

Sheffield Hallam University

A transputer based parallel database system.

GRAY, Jonathan Patrick.

Available from the Sheffield Hallam University Research Archive (SHURA) at:

<http://shura.shu.ac.uk/19717/>

A Sheffield Hallam University thesis

This thesis is protected by copyright which belongs to the author.


The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Please visit <http://shura.shu.ac.uk/19717/> and <http://shura.shu.ac.uk/information.html> for further details about copyright and re-use permissions.

~~POLYTECHNIC LIBRARY~~
~~178P~~
SHEFFIELD S1 1WB

100 312 436 4



13305

Sheffield City Polytechnic Library

~~23702~~

REFERENCE ONLY

~~23718~~

Fines are charged at 50p per hour

28 .III. 2004

4.14pm

ProQuest Number: 10697019

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 10697019

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

A Transputer Based Parallel Database System

by

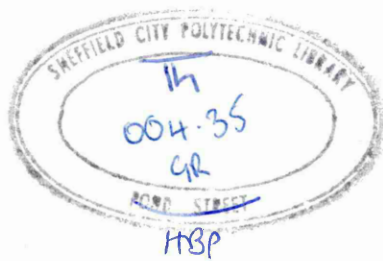
Jonathan Patrick Gray

A thesis submitted to the Council for National Academic Awards in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Sponsoring Establishment: School of Computing and
Management Sciences,
Sheffield City Polytechnic

Collaborating Establishment: National Transputer Support Centre,
Sheffield

January 1991



HBP

Abstract

A sophisticated database application generation environment known as DB4GL has been developed at Sheffield City Polytechnic. A unique feature of DB4GL is the object-oriented application model used to specify and generate database applications. Although DB4GL has many advanced and powerful features, such as a self-describing data dictionary and extensive integrity rule processing facilities; the system has not been designed for high performance in either the generation tools or the generated database applications. The Parallel-DB4GL (P-DB4GL) project represents an attempt to improve the performance of the generated database applications, by constructing a new concurrent implementation of DB4GL for execution on transputer-based parallel hardware.

This thesis describes the DB4GL system as developed to the commencement of the P-DB4GL project. A prototype P-DB4GL system has been implemented that demonstrates how significant performance gains can be obtained from a concurrent implementation on transputer-based parallel hardware. Based on the successful results of this prototype system, designs for a fully functional multiprocessor P-DB4GL system are proposed. The details of this prototype and the fully functional designs are presented in this thesis. The thesis also provides an evaluation of the P-DB4GL project as a whole, and concludes with some suggestions for further research in the areas of parallel databases and object-oriented system implementation.

Acknowledgements

I would like to take this opportunity to thank my supervisor, Professor Frank Poole, for the excellent supervision and encouragement he has given. In addition I would also like to thank Mrs Innes Jelly for her valuable comments in the preparation of this thesis and other reports. Finally I would like to thank Ms Susan Cleary for her assistance in typing, proof reading, and printing of this thesis.

Contents

Chapter 1

Introduction

1.1 Motivation for the Research.....	1
1.2 Aims of the Research	2
1.3 The P-DB4GL Approach.....	3
1.4 P-DB4GL Project Development.....	5
1.5 Outline of Thesis Contents	6

Chapter 2

The DB4GL Database Application Generation System

2.1 DB4GL Project Overview.....	10
2.2 Application Generation and Prototyping	10
2.3 DB4GL Application Development	11
2.4 The DB4GL Application Model.....	12
2.4.1 A Generic Architecture.....	12
2.4.2 Presentation Objects.....	13
2.4.3 Information Units.....	14
2.4.4 Data Relationships	15
2.4.5 Levels of Data Description.....	16
2.5 Implementation of the Application Model.....	16

Chapter 3

Object-Oriented Database Approaches

3.1 A Classification of Object-Oriented Approaches	30
3.2 Object-Oriented Approaches and DB4GL.....	32

Chapter 4

Transputers, Occam, and Databases

4.1 Transputers.....	38
4.2 Occam.....	38
4.3 Transputers and Databases.....	39 ←
4.4 Project Resources.....	42

Chapter 5

Developing the Prototype Parallel-DB4GL

5.1 Project Objectives	45
5.2 Design Methodology.....	45
5.3 Development Approach.....	46

5.4 Implementation	47
5.5 Testing	51
5.6 Results	53
5.7 Evaluation	54

Chapter 6

Designs for a Fully Functional P-DB4GL

6.1 A Multiprocessor Fully Functional P-DB4GL.....	67
6.2 Channel Multiplexors and Message Routers.....	68
6.2.1 Channel Multiplexing.....	68
6.2.2 P-DB4GL Channel Multiplexors	69
6.2.3 P-DB4GL Message Routing.....	70
6.3 Communication Loads and Protocol Overheads.....	72
6.4 Designs for P-DB4GL Hardware Configurations	75
6.5 Code Factoring	77
6.6 Code Generation.....	78

Chapter 7

P-DB4GL Project Evaluation

7.1 Summary	90
7.2 Communication Loads and Object Clustering.....	90
7.3 Object Inheritance	91
7.4 Occam 2 as an Implementation Language.....	91
7.5 Improvements in Secondary Storage Technology.....	92
7.6 The Next Generation of Transputer Products	93
7.7 Advantages of a Prototyping Development Cycle	95

Chapter 8

Conclusions..... 97

Bibliography..... 99

Appendices

Appendix A - Algorithm Syntax Definition	110
Appendix B - Message Formats and Occam Channels.....	111
Appendix C - Entity and Schema Handler Algorithms.....	115
Appendix D - Disc and Filer Algorithms	126
Appendix E - Filer Harness Specification.....	130
Appendix F - User Process Test Harness	138
Appendix G - Test Data and Test Configurations.....	144

Appendix H - Test Results.....	151
Appendix I - Test Application Performance Optimization.....	167
Appendix J - Routers and Multiplexors	176
Appendix K - System Constants.....	192
Appendix L - Channel Protocols.....	195

List of Figures

Figure 1.1 - Thesis Contents.....	9
Figure 2.1 - DB4GL Development Environment.....	18
Figure 2.2 - Information for Application Specification.....	19
Figure 2.3 - DB4GL Application Model.....	20
Figure 2.4 - Database Applications as Instantiations.....	21
Figure 2.5 - Examples of Application Structures	22
Figure 2.6 - Data Dependencies and Object Activity	23
Figure 2.7 - Prime View and Non-prime View Information Units.....	23
Figure 2.8 - Domains and Relationships.....	24
Figure 2.9 - Information Unit Groups and Schema Links	24
Figure 2.10 - ANSI/SPARC DBMS Model	25
Figure 2.11 - Levels of DB4GL Data Description	26
Figure 2.12 - Implementation of the Application Model	27
Figure 2.13 - Generated Application Instances.....	28
Figure 2.14 - DB4GL Code Generation	29
Figure 2.15 - Modular Structure of Implemented Applications	29
Figure 3.1 - An Object-Oriented Design.....	36
Figure 3.2 - DBMS/Object-Oriented Environment Interface.....	37
Figure 4.1 - Transputer Architecture.....	43
Figure 4.2 - The Occam Language	44
Figure 5.1 - System Configuration for P-DB4GL.....	58
Figure 5.2 - Concurrent Message Passing Processes	58
Figure 5.3 - Typical P-DB4GL Database Application.....	59
Figure 5.4 - DB4GL and P-DB4GL Message Passing.....	59
Figure 5.5 - Concurrent Execution of Data Access Processes.....	60
Figure 5.6 - Developing the Prototype P-DB4GL System	61
Figure 5.7 - Mapping Entity Handlers to Discs.....	62
Figure 5.8 - Typical Test Application Configuration	63
Figure 5.9 - Version 4 Handlers Test Run Times.....	64
Figure 5.10 - Normalized Coupling Entity Update.....	64
Figure 5.11 - Version 4 Handlers Improvement Factor.....	65
Figure 5.12 - Test Run Processor Loading	65
Figure 5.13 - Processing/Disc Access Ratio.....	66
Figure 5.14 - Processor Loading and Parallel Configuration.....	66
Figure 6.1 - Insufficient Links for Filer Connection	80
Figure 6.2 - Distributing the Data Access Code.....	80
Figure 6.3 - Designs for a Fully Functional Multiprocessor P-DB4GL.....	81
Figure 6.4 - Configuration Mismatch Between Channels and Links.....	82

Figure 6.5 - Addition of Multiplexor Processes.....	83
Figure 6.6 - Modified Prime Entity Handler.....	83
Figure 6.7 - P-DB4GL Decoder Process.....	84
Figure 6.8 - Routing Messages Through a P-DB4GL Network.....	84
→ Figure 6.9 - Different Types of P-DB4GL Messages.....	85
Figure 6.10 - Multiplexing Filer Channels.....	85
Figure 6.11 - A Separate Interconnection Network.....	86
Figure 6.12 - A Reconfigurable Network.....	87
Figure 6.13 - Replacement of Schema Handler by IUG Process.....	88
Figure 6.14 - An Alternative Presentation Object Implementation.....	88
Figure 6.15 - Methods of Code Generation.....	89
Figure 7.1 - An H1 Based P-DB4GL Design.....	96
Figure B1 - P-DB4GL Message Formats.....	114
Figure E1 - Filer Harness Record Representation.....	136
Figure F1 - A Single Transaction.....	140
Figure F2 - Optimised Sequence of Transactions.....	141
Figure F3 - A Sequence of BCU Messages.....	142
Figure F4 - Effect of User Harness Delays.....	143
Figure G1 - Test Entities (Information Units).....	147
Figure G2 - Test Schemas.....	148
Figure G3 - Test Data Access Code.....	149
Figure G4 - Test Configurations.....	150
Figure I1 - Effect of Message Size on Transfer Rate.....	174
Figure I2 - Channel Protocol Converter Processes.....	174
Figure I3 - Protocol Conversion Test Configuration.....	175
Figure J1 - P-DB4GL Router Process.....	188
Figure J2 - Network8 Ring Router Process.....	188
Figure J3 - Network9 Ring Router Process.....	189
Figure J4 - Net16 Ring Routing Test Configuration.....	190
Figure J5 - Net18 Ring Routing Test Configuration.....	190
Figure J6 - Net19 Ring Routing Test Configuration.....	191

Chapter 1

Introduction

1.1 Motivation for the Research

This PhD thesis describes the work undertaken and results obtained from the Parallel DB4GL (P-DB4GL) research project. The P-DB4GL project is an extension of continuing research at Sheffield City Polytechnic in the areas of databases and automated systems development. During recent years several research projects concerned with data dictionaries, 4th Generation Languages (4GL's), and application prototyping have been undertaken at the Polytechnic [Cooper83] [Beazley84] [Ewin84] [Bal85] [Priti86] [Kilo86]]; culminating in the development of a sophisticated database application generation environment known as DB4GL (Data Base 4th Generation Language) [Ewin85a] [Ewin85b] [Poole87] [Hird89].

The DB4GL research was motivated chiefly by concerns of improving programmer productivity and automating software development. DB4GL, like many 4GL's, exists as a collection of productivity enhancing tools [Nelson85]. These tools, such as data dictionaries, normalization engines, screen painters, report generators, and code generators, are used by system analyst/designers and programmers to improve efficiency and shorten development time in the traditional software system development cycle, or "waterfall" life cycle [Boehm76], of analysis-specification-design-implementation-testing [Cobb85] [Forage85]. Such 4GL tools can also be used to rapidly build, or "prototype", applications in a modified software development cycle [Brittan80] [McCraken82] [Appleton83] [Johnson83].

DB4GL represents an attempt to create a unifying ordered approach to the construction of database-oriented information systems (or database applications) within the context of a prototyping system development methodology. DB4GL integrates both processing and information descriptions, combined with rule-based integrity constraints, in a formal homogeneous object-oriented structure known as the DB4GL application model. This application model is fundamental to the generation of database applications; it is a generic structure, or template, from which generated applications are derived. Generated database applications are defined as specific instances of this generic structure.

The DB4GL generated database applications, and the DB4GL tools themselves, have not been designed with performance in mind. Some of the DB4GL tools, the generator tools in particular, are very slow in operation. The compiled program code that constitutes the generated DB4GL applications is slow in execution, and could have been more efficiently designed and coded. Because DB4GL has been developed as a research vehicle rather than a commercial product, performance issues have not been a major concern. Fortunately, DB4GL is fully portable to any system supporting a COBOL compiler, and many of the performance problems can be solved by transferring to target hardware of a higher specification than the PC system currently used. However, even with more efficient coding and faster hardware, there is a limit to the improvements that can be obtained with the existing DB4GL implementation.

The motivation for the P-DB4GL project is performance. To try and solve the performance problems encountered in the sequential COBOL implementation of DB4GL, a radically different approach has been taken, that is, the construction of an entirely new parallel implementation of DB4GL.

1.2 Aims of the Research

The aim of the P-DB4GL project is to redesign the DB4GL generated database applications so that they can be executed on parallel hardware. The potentially massive processing power of this parallel hardware can then be used to speed-up the performance of the generated applications, and also allow the applications to be scaled-up to much larger database sizes without loss of performance. The DB4GL application model is retained. This has been shown to be a suitable software architecture for application specification and generation, and the P-DB4GL project is a test of the application model's suitability for parallel implementation. In particular, the object-oriented aspects of the application model are examined to see if an object-oriented specification facilitates parallel implementation.

The parallel hardware chosen for the P-DB4GL implementation is transputer based. Transputers [Inmos88a] are a family of VLSI devices, including RISC (Reduced Instruction Set Computer) [Bell86] [Gimarc87] microprocessors and peripheral controllers, specially designed for the construction of distributed memory MIMD (Multiple Instruction Multiple

Data) [Flynn72] architectures. Each transputer microprocessor is a "computer on a chip" and combines on a single integrated circuit, CPU, RAM and high speed serial communication links. Transputers are relatively cheap and are easily connected together by their serial communications links to produce large extensible parallel architectures. The low cost and extensibility of transputers, combined with the availability of several high level parallel languages, makes them suitable candidates for the parallel hardware. Also, at the start of the P-DB4GL project, it was envisaged that the parallel DB4GL database applications would be essentially "shared nothing" architectures [Stonebraker86], and the transputer's built-in communication facilities were considered appropriate for the construction of such architectures.

1.3 The P-DB4GL Approach

The P-DB4GL project differs significantly from many other research projects concerned with database performance issues. In particular, there are three aspects to the P-DB4GL approach which differentiate it from other database approaches. First, it differs architecturally from established database machine approaches. Secondly, the DB4GL application model represents a novel approach to database application development. Thirdly, P-DB4GL is intended for use in a prototyping development environment, and this offers certain advantages; in particular, it facilitates performance "tuning" of the generated database applications.

The P-DB4GL project is not an attempt to construct a dedicated "backend" database machine [Hsiao83]. Typically, such database machines are constructed from specialized custom hardware, and are invariably concerned with improved disc access and file processing, for example: CAFS [Babb79], DIRECT [DeWitt79], SABRE [Gardarin83], RDBM [Schweppes83], DBMAC [Missikof83], Gamma [Schneider89]. They are generally not stand-alone machines, but dependent on either a host machine (usually a mainframe computer) in which they replace some of the operating system's functions and execute DBMS code more efficiently than the host machine; or else function as a database server forming part of a networked or distributed system, receiving database queries (typically in SQL [BSI88]) from client processors, processing these requests and returning results of this query to the clients. DB4GL differs from such database machines in that all of the database application executes entirely on the transputer-based hardware, there are no separate client or host processes being serviced by P-DB4GL.

Secondly, P-DB4GL applications are fully distributed parallel applications, and exhibit inter-query parallelism which arises from the inherently concurrent specification of DB4GL applications.

A unique feature of the P-DB4GL research is the DB4GL application model. This application model represents an attempt to specify and generate database applications in their entirety. It encompasses all facets of a database-oriented information system - that is, processing, data model, user interface, and integrity constraints - within a uniform structure. Thus, DB4GL differs from other database approaches, in that there is no distinction between DBMS functions and separate application programs using the DBMS. The application model is not relational, but has certain object-oriented characteristics. Specifically, a database application is defined by the model as a set of concurrently executable message passing Presentation Objects, and each particular generated application is considered as an instantiation of the application model. DB4GL is further distinguished from some other database approaches by the special emphasis placed upon the attribute, rather than tuple or relation, as the primary data item.

The use of DB4GL within a prototyping development environment has a significant effect on some implementation features. DB4GL applications are oriented towards on-line transaction processing (OLTP) rather than on-line query processing. This means that all the access paths (explicit data relationships) and patterns of data usage are known and defined in advance of end-user operation of the generated application. Consequently certain optimizations to data access mechanisms, such as the establishment of indexes on frequently used tables, are possible. Indexes reduce search time and improve response time to transactions, however, one disadvantage of indexes is the processing overhead entailed by the maintenance of indexes following an update to the table over which they are defined. The use of parallel processing hardware to improve index maintenance is a key feature of the P-DB4GL system.

A possible solution to the problem of providing an ad hoc query processing facility for a DB4GL system (assuming an appropriate DB4GL application does not already exist to satisfy a given query) is the rapid building of a specific (OLTP-oriented) application for that query. However, with the existing slow DB4GL generator tools this can take several minutes, and hardly constitutes on-line ad hoc query processing. An advantage of a faster

P-DB4GL running on parallel hardware is the potential for greatly reduced application generation time, thereby improving the query processing facilities of DB4GL.

The prototyping development cycle facilitates the "tuning" of P-DB4GL applications for optimal performance. A problem common to many parallel database implementations is efficient resource utilization, that is, matching processing and communication loads of the database software to the available hardware. Because P-DB4GL development is cyclic, at each cycle in the application development, the system analyst/designer has the opportunity of altering the mapping of application code and database data to the available processors and secondary storage devices (disks). Thus allowing the possibility of experimentation with different configurations to find a well balanced configuration offering optimal performance.

1.4 P-DB4GL Project Development

The first stage in the parallel DB4GL investigation was the development of a prototype P-DB4GL system. In this prototype system, simplified "cut-down" database applications were designed and implemented. These P-DB4GL test applications are of reduced functionality, and are equivalent to applications generated by the original [Ewin85a] DB4GL. In a fully functional P-DB4GL, the parallel database applications would be generated automatically by P-DB4GL generation tools, but these tools are not yet fully implemented, and in the prototype system the test applications were hand coded. The test applications were run on several different network configurations, and designs for a fully functional P-DB4GL system are proposed based on the results obtained from these tests.

The reasons for constructing a prototype P-DB4GL are:

- determine the exact amount of parallelism inherent in the DB4GL application model;
- test the soundness of the design approach, that is, of a relatively simple translation from an object-oriented specification to an implementation of concurrent communicating processes;
- assess the suitability of Occam as an implementation language for P-DB4GL;

- examine the feasibility of using transputers as the basis of the parallel processing hardware, both for P-DB4GL in particular and other database (and data intensive) systems in general.

Briefly, the main findings from the prototype P-DB4GL investigation are:

- significant performance benefits were obtained from parallel index maintenance;
- speed-up of any given application by the simple addition of more processors is very difficult to obtain;
- good scale-up in performance by the addition of more processors and discs can be obtained as an application grows larger;
- the P-DB4GL database applications do not significantly load the processors, but applications often become communication bound.

Based on these findings, some design considerations for a fully functional P-DB4GL system are discussed in Chapter 6.

1.5 Outline of Thesis Contents

The P-DB4GL research project, as described in this thesis, brings together a number of strands, or themes, of different research interests (Figure 1.1). One of the core themes, providing motivation and background to the DB4GL project, concerns database application generation and prototyping software development. Another core theme is concerned with the technology of parallel processing hardware, specifically transputers and Occam, and the application of transputer technology to data intensive application areas. Parallel architectures for databases and knowledge based systems, including dedicated database machines, specialized relational and logic language machines, and hardware support for object-oriented environments, is a third theme of research. Finally, a concern with software design and development methodologies for concurrent systems completes the picture.

These four different themes of research interest are all components in the P-DB4GL project, and are present in various chapters of this thesis. In the following description of each chapter's contents, the relevance and influence of these themes is indicated.

Chapter 1, the Introduction, introduces the research themes, and explains the motivation and aims of the P-DB4GL project.

Chapter 2 describes the DB4GL Database Application Generation System. DB4GL is the result of research encompassed by the theme of database application generation. Chapter 2 describes the DB4GL system as developed by [Hird89], and pays particular attention to the DB4GL application model which has been greatly influenced by developments in object-oriented research.

Chapter 3, Object-Oriented Database Approaches, provides a simple classification of object-oriented approaches to database development. This classification is used to provide a background, or framework, in which the influence of object-oriented concepts on DB4GL is explained. DB4GL is not representative of the "mainstream" Object-Oriented Database Systems, but the development of DB4GL has drawn heavily on research developments in the object-oriented fields.

Chapter 4, Transputers, Occam and Databases, provides a short introduction to the transputer technology, and the concurrent programming language Occam, used in the construction of the parallel architecture for P-DB4GL. Other database research projects, also using transputer-based technology, are described; and the advantages of transputers for parallel database implementations is explained.

Chapter 5 describes the development of the Prototype Parallel-DB4GL system. The design, implementation, and testing of the prototype P-DB4GL implementation is described in detail. In particular, the design approach, a translation from concurrent object specification to an implementation of communicating concurrent Occam processes, is explained. Based on the results obtained from this prototype system, designs for a fully functional P-DB4GL system are proposed.

Chapter 6, Designs for a Fully Functional P-DB4GL, describes design proposals for a multiprocessor fully functional P-DB4GL system. Some features of these designs, such as channel multiplexing and message routing, have been implemented and tested, and in the light of these results, revisions to the designs are made.

Chapter 7 provides an Evaluation of the P-DB4GL Project, including both the prototype P-DB4GL system and the fully functional designs, and indicates where opportunities for further research lie.

Chapter 8 provides some overall Conclusions from the P-DB4GL project.

Lastly, there is Bibliography of all works cited in the thesis. Followed by a number of Appendices, which mostly contain very detailed information about test data, test results, and algorithms; this has been removed from the main text to make it more concise and readable.

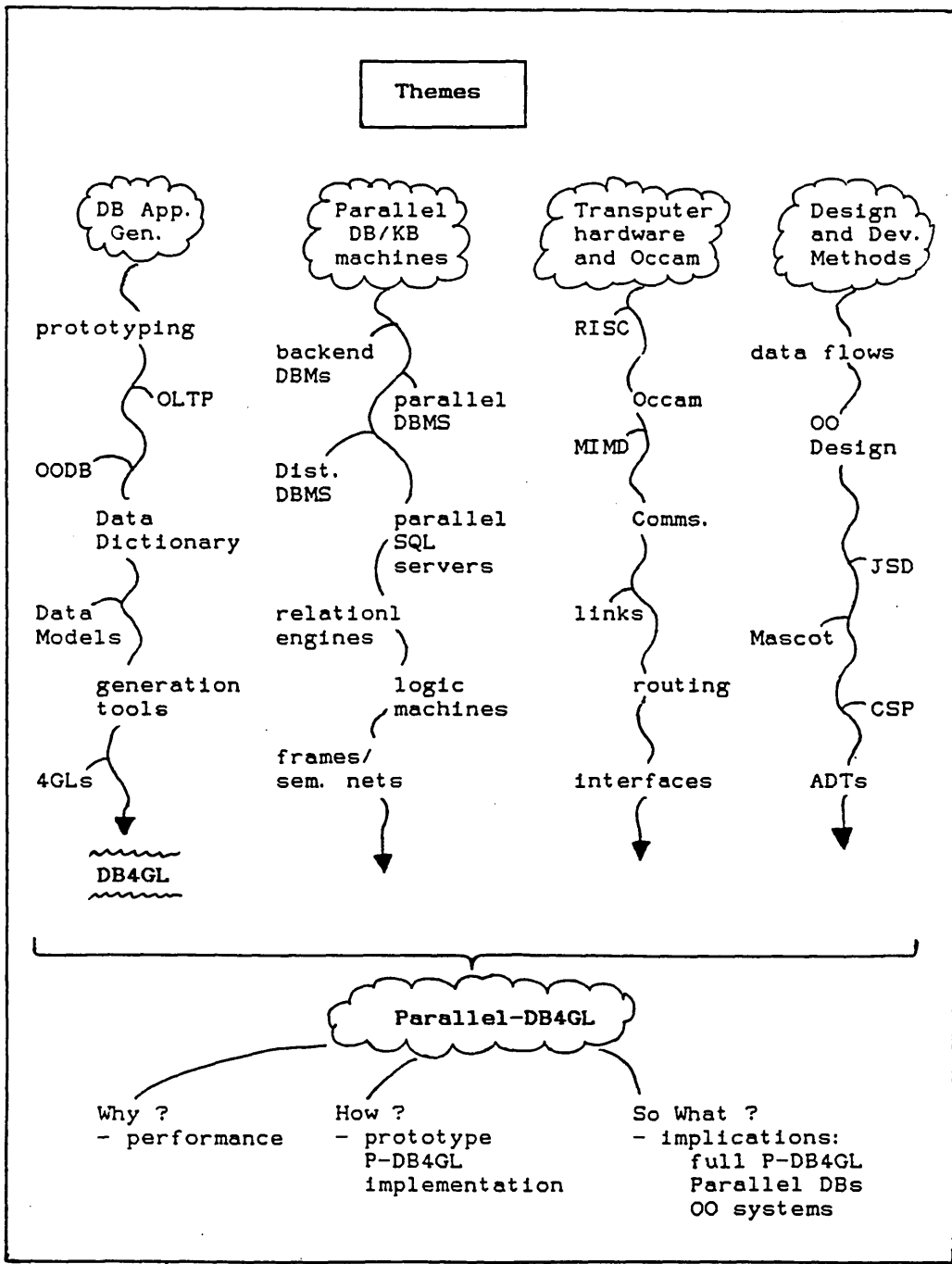


Figure 1.1 - Thesis Contents

Chapter 2

The DB4GL Database Application Generation System

2.1 DB4GL Project Overview

DB4GL (Data Base 4th Generation Language) [Ewin85a] [Ewin85b] [Hird89] [Poole87] is a database application generation environment that has been developed at Sheffield City Polytechnic. It is a result of research conducted in the Department of Computer Studies (now incorporated into the School of Computing and Management Sciences) in the areas of database application prototyping and Fourth Generation Languages (4GL's). DB4GL is intended for use within a rapid system building (or prototyping) development cycle [Appleton83] [Dearnley83] [McCracken82]. DB4GL is designed primarily for the generation of on-line transaction processing systems, rather than on-line enquiry systems.

Although DB4GL has been developed principally as a research vehicle, it has been applied to commercial problems. The first version of DB4GL [Ewin85a] was used successfully to generate database applications for a construction company [Ewin84] [Ewin85a]. Subsequent DB4GL research [Poole87] [Hird89], concentrating in the areas of improved specification tools, integrity rule processing, and data dictionary operation, has transformed the original DB4GL system into a sophisticated 4GL environment with a unique object-oriented application model.

2.2 Application Generation and Prototyping

The term "prototyping", when used in the context of software development, means rapidly building an implementation for trial purposes. This provides users with a (sometimes partially) working model of the proposed system in order to identify and define their requirements more precisely. There are two main forms of prototyping:

- "throw-away" requirements analysis prototyping, whereby the prototype systems are discarded once user requirements are fully established;
- "evolutionary" prototyping, in which successive prototypes are developed and the final prototype becomes the delivered system.

Typically, 4GL's (fourth generation languages) or "application generators" [Horowitz85] [Luker86] are used for prototyping.

An application generator is a piece of software (more normally a suite of software tools) that produces a program in some object language, which is tailor-made to fit a specific set of requirements. Application generators (or 4GL's) differ from general purpose high level language translators in that:

- the target language is usually another high level language;
- the range of programs that can be generated, is limited to a particular application domain, for example, database applications;
- often, input to a 4GL does not have to conform to rigid rules of syntax, input is via a dialogue with the user, who may respond in some free format;
- the user of 4GL may not be a professional programmer, he may be an end user or system analyst/designer.

2.3 DB4GL Application Development

DB4GL is an application generation system specifically for database applications. DB4GL generates single user database applications to run on an IBM PC AT microcomputer. The target language generated by DB4GL is COBOL. The specifications of required applications are input to DB4GL using menu-driven screen-based database maintenance applications. DB4GL can be used for either "throw-away" prototyping or for "evolutionary" prototyping. The DB4GL system is intended to be used principally by a system analyst/designer rather than an application end user.

DB4GL is a collection of tools - report generators, screen painters, code generators - based around a data dictionary. The DB4GL data dictionary is active, in that, applications are generated from the specifications stored in the data dictionary at generation time, the data dictionary does not just passively describe applications. The DB4GL data dictionary is itself maintained by DB4GL generated applications and uses a hybrid self-describing [Mark85] [Mark86] [Roussopoulos85] data model [Hird89] [Poole87], originally based on the Entity-Relationship approach [Chen76] [Howe83], but also influenced by other data models, for instance, DIAM [Senko73] [Senko76] and CODASYL [Olle78].

The DB4GL application development cycle is illustrated in Figure 2.1: the specification/generation/user-feedback cycle iterates until the system designer and application user are satisfied with the generated database

application. DB4GL is used mainly for "evolutionary" prototyping, in which the final generated prototype is the delivered database system. The delivered database system can then be operated by the application user without further DB4GL support.

Figure 2.2 illustrates the types of information that constitute the specification of a DB4GL database application, it includes:

- **data model description**, in terms of entities, attributes, domains, and schema links (access paths);
- **processing requirements**, in terms of standard maintenance operations and query and report producing functions defined over the entities and attributes of the application;
- **user interface**, in terms of screens and windows of input and output data requested and produced by the processes specified above.

Often, considerable data analysis skills are needed, particularly in the case of large complex applications, in order to properly specify the required database application. Such skills are not normally possessed by application end users. It is therefore unlikely that an application end user would be able to define and generate a DB4GL database application without the help of a skilled system analyst/designer.

2.4 The DB4GL Application Model

2.4.1 A Generic Architecture

The DB4GL application model, or "architecture of a database application", is central to the generation of database applications (Figure 2.3). This application model is a generic structure or template from which particular application instances are derived. The DB4GL generation tools use the specifications stored in the DB4GL data dictionary to produce a required database application that is an instantiation of the generic DB4GL application model (Figure 2.4). The key features of the application model are:

- an application task is defined as a set of Presentation Objects (PO's);
- each constituent PO of an application task includes both information entities and their associated processing tasks;
- each PO is, in principle, capable of concurrent execution, and only interacts with other PO's in the application task by

- communicating control/data messages;
- during the execution of an application task, the course of computation is message driven.

In the simplest of database application, all the required data and processing can be encompassed in a single application task. It is, however, more usual for a DB4GL database application to be constructed as a suite of application tasks. The application tasks are linked together by a menu-driven selection program, produced by a DB4GL menu generator tool. The menu program is used by the application user to select a database processing option, and the menu program invokes the appropriate application task.

2.4.2 Presentation Objects

The concept of the Presentation Object (PO), and its role in the Application Model (AT), is of fundamental importance to DB4GL. Essentially, a Presentation Object combines descriptions of:

- persistent database data;
- processing operations, with their associated transient data, performed upon this persistent data;
- integrity rules, controlling updates to the persistent data.

The definition of an AT partitions a database into a number of disjoint, though related, parts; and each PO operates upon one of these separate parts of the database. The execution model is one of co-operating non-interfering objects.

Specifically, each PO is composed of:

- a Presentation Unit (PU), which defines the sequence of data access and program control required to support the user interaction for the PO;
- an Information Unit Group (IUG), that is, a collection of related data entities, known as Information Units (IU's), processed by the PO;
- a Process Schema (PS), with a number of Process Tasks (PT's) for processing the IUG, the PT's may be functions performing simple data type conversions, or they can be complex processes generating aggregate values and derived data from the IUG;
- a Presentation Format (PF), describing both the appearance of the PO to application users and the interface to other PO's within an

application;

- a set of Integrity Rules controlling updates to the IUG, some of these may be Generic rules applicable to all application tasks, others may be application dependent rules specific to this application task.

When a Presentation Object (PO) is activated, its Presentation Unit (PU) controls the PO's user interaction. The PU provides a dialogue between the PO and application user, whereby the application user can effect a range of query and maintenance operations upon the data defined in the PO. Typically, these are operations to select, insert, modify, and delete particular entities and attributes in the PO's IUG. The user dialogue includes commands to suspend the PO's activity and transfer user interaction to another PO within the application task.

The component Presentation Objects (PO's) in a DB4GL application are, in principle, capable of concurrent execution. However, relationships between PO's, based on data dependencies between the data entities defined within each PO, determine the order in which PO's are activated during application execution. This ordering of PO activation during application execution is known as an application structure, and a notation for expressing this application structure has been developed in [Hird89]. Many different application structures can be defined using this notation, Figure 2.5 illustrates some of these. Figure 2.6 provides an example of the use of this notation to define a simple application structure composed of just two PO's. It illustrates how a sequence of PO activation is imposed by a relationship between the ORDER entity, defined in superordinate PO1, and the dependent ORDER-LINE entity, defined in subordinate PO2.

2.4.3 Information Units

In DB4GL, the attribute is considered to be the primary data item. However, to support efficient storage and retrieval of data from backing store, attributes are aggregated into data entities known as Information Units (IU's). An IU is a collection of attributes functionally dependent on a primary key. An IU identifies some unit of information in the application domain. All access to IU occurrences is via the IU primary key. There are two types of IU: **prime view IU's**; and **non-prime view IU's** (Figure 2.7). Each prime view IU may have many non-prime view IU's associated with

it. A non-prime view IU permits access to its prime view IU occurrences using a non-key attribute, and is formed from a concatenation of the non-key attribute with the primary key of the prime view IU. A non-prime view IU is said to be closely coupled to its prime view IU, in that, any updates to the prime IU must also be accompanied by corresponding updates to all the associated non-prime IU's. At the physical storage level, each IU is realised by a single relation implemented as an Indexed-Sequential file of records and a processing object known as an entity handler. Prime view IU's have a prime entity handler, non-prime IU's have a coupling entity handler. Thus, a non-prime view IU's coupling entity handler can be seen as maintaining a kind of "index" to a prime IU's "base relation".

2.4.4 Data Relationships

All attributes are defined over domains. These domains specify the range and type of possible attribute values, eg 4 digit integer, 80 character alphanumeric, 5 digit fixed point with 2 d.p., etc. Relationships between Information Units (IU's) are established on the basis of their domain related attributes. Figure 2.8 illustrates one-to-many relationships between ORDER, ORD-LINE, and PART IU's, established through the common domains 0# and P#.

When application tasks are defined, relationships between IU's are made explicit through the specification of schema links. Schema links are represented by unidirectional arcs (Figure 2.9). Each link has a source attribute, which may be key or non-key, and a target attribute, which must be a key attribute. Within an IUG, a schema link denotes a one-to-one relationship between the connected IU's. Between IUG's, a schema link normally denotes one-to-many relationships.

For a given application task, the set of all IU's contained in the IUG's of the constituent PO's, combined with all inter- and intra- IUG schema links, is collectively known as the application task's data access schema. In the implementation of DB4GL applications, a processing object known as a schema handler controls access to the IU's of a schema. The schema handler uses the schema links to prefetch (or realise) related IU occurrences (records) from the filestore.

2.4.5 Levels of Data Description

DB4GL contains a number of levels of data description or data abstraction. In terms of the ANSI/SPARC [Jardine77] [Tsichritzis78] DBMS reference model (Figure 2.10), it is possible to identify the three corresponding levels of data description in DB4GL (Figure 2.11). The DB4GL conceptual level is represented by the set of all the information units (IU's) defined in a DB4GL system, together with the complete set of generic integrity rules defined over these IU's. Explicit data relationships between IU's, defined by schema links, are not represented at the conceptual level. The definition of application tasks with their presentation objects, information unit groups, schema links, and application specific integrity rules belong to the external level. The indexed-sequential files (tables of records), used to implement the IU's, correspond to the physical level.

2.5 Implementation of the Application Model

The DB4GL application model is implemented by a number of generic code modules (Figure 2.12). These modules can be broadly classified into:

- **User Modules** (Screen, Window, and Process modules) that process user requests and send commands to the data access schemas, and;
- **Data Access Modules** (Schema handler and Entity handler modules) that retrieve and store persistent application data in the filestore.

Figure 2.12 shows the mapping from the application model to the code modules. The user modules implement the presentation units, process schemas and process tasks of an application task's presentation objects. Each information unit in an application task is supported by an entity handler module. A schema handler module supports the application task's data access schema.

When a database application is generated, the required application is specified as an instance of the DB4GL application model. The specification is in terms of presentation objects with IU's, IUG's, schema links, process schemas and process tasks. It is in this form that the specification is entered into and stored in the DB4GL data dictionary. The DB4GL generation tools take this specification stored in the data dictionary, and, using the generic code modules, produce instantiated code modules which constitute the

implemented application instance (Figure 2.13). The generic code modules are in fact skeleton COBOL programs. The instantiated code modules (completed source code) then have to be compiled (Figure 2.14). The implemented application instance is executed as a collection of separately compiled COBOL modules linked together by an overall hierarchical control structure. Data and control messages are passed as parameters at code module calls (Figure 2.15).

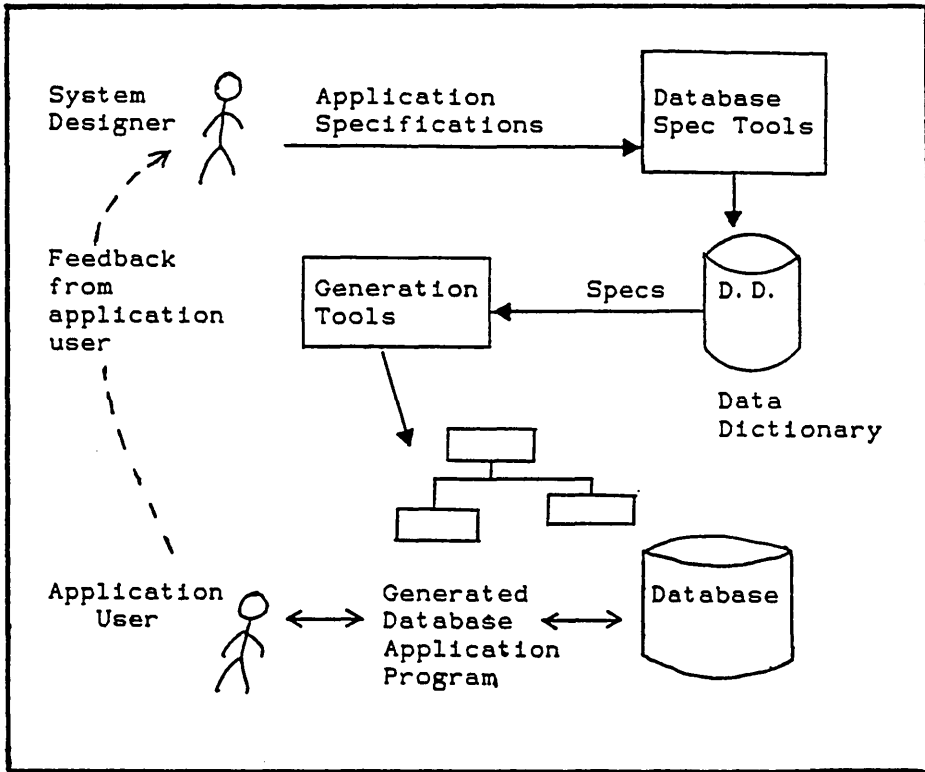


Figure 2.1 - DB4GL Development Environment

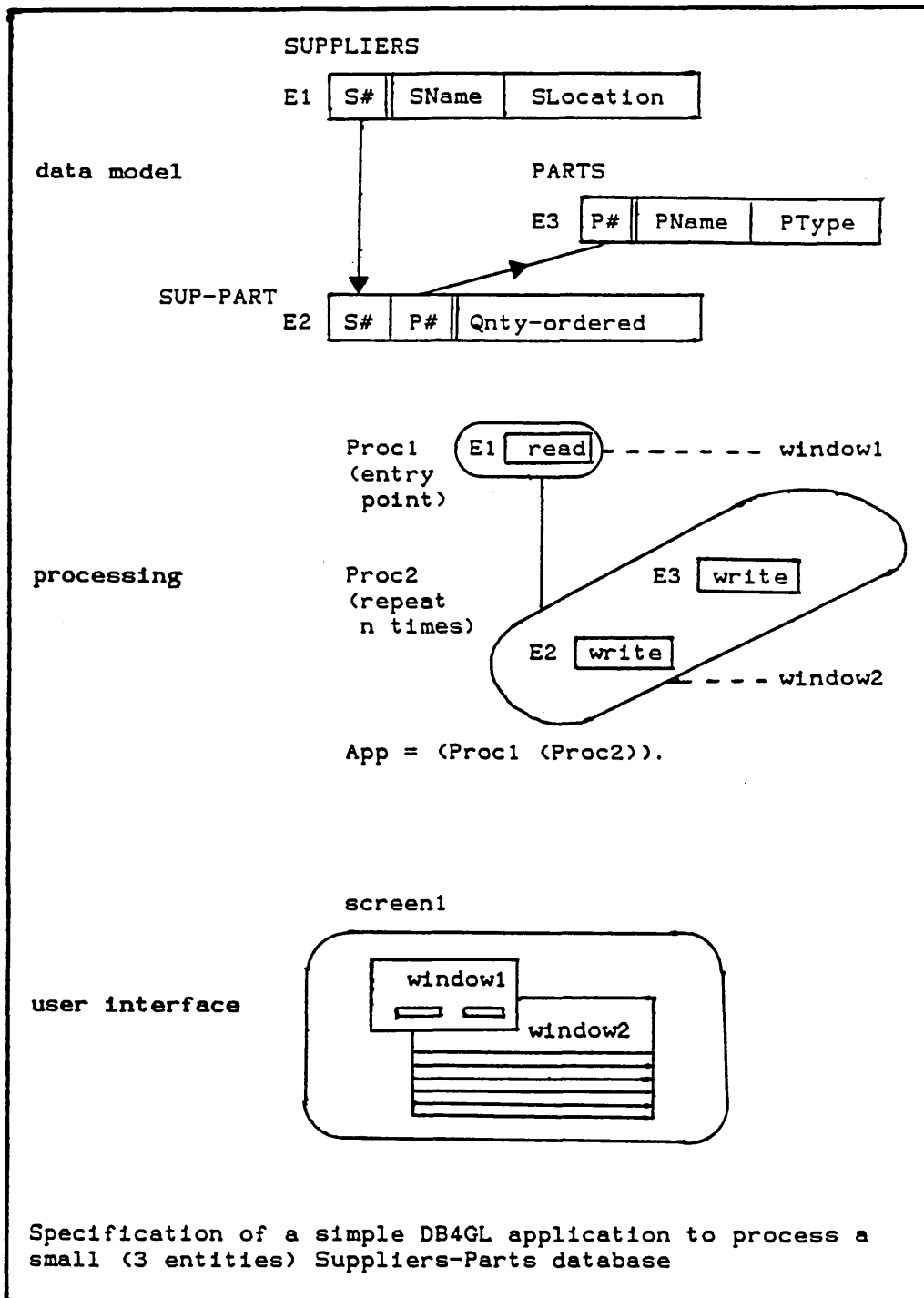


Figure 2.2 - Information for Application Specification

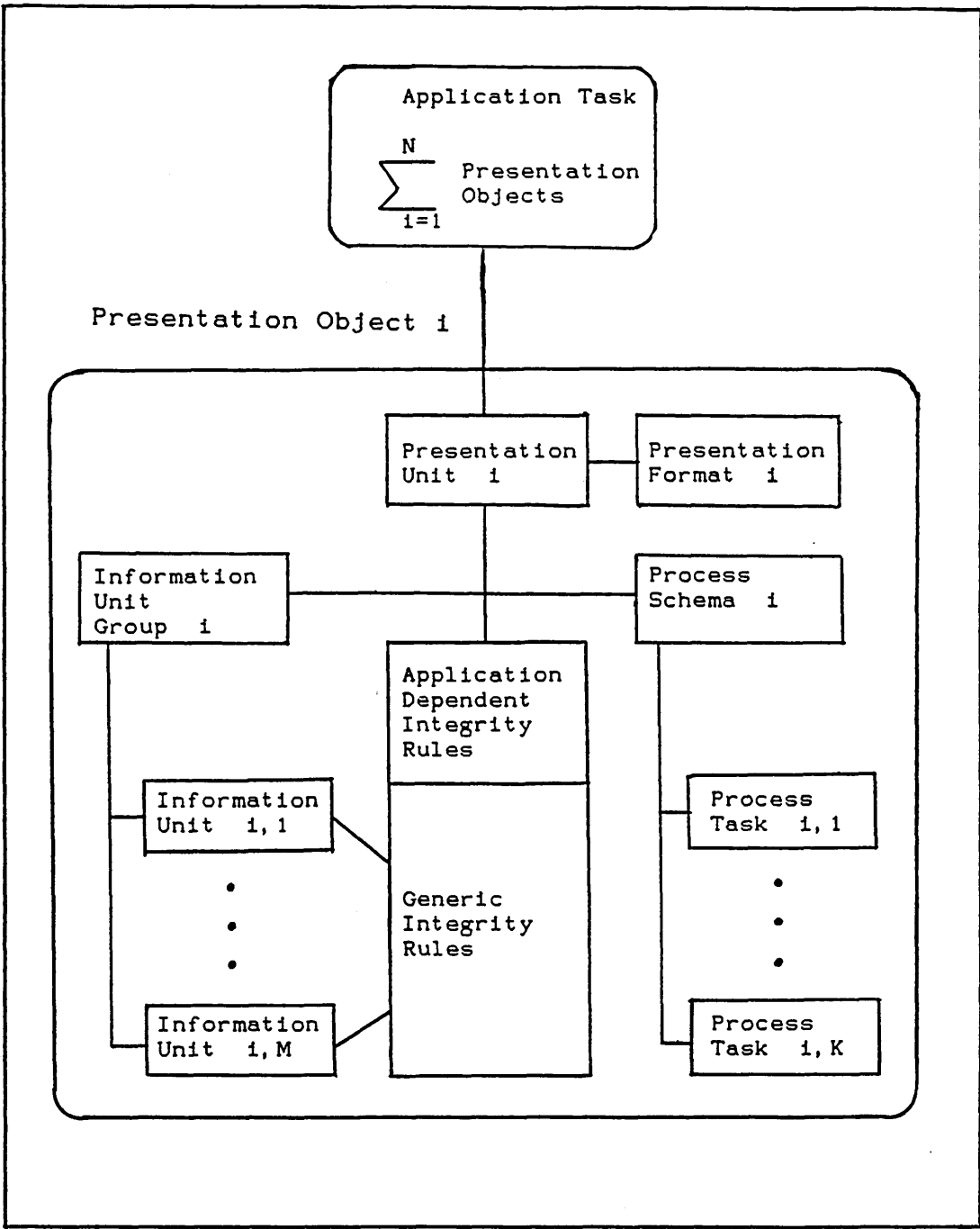


Figure 2.3 - DB4GL Application Model

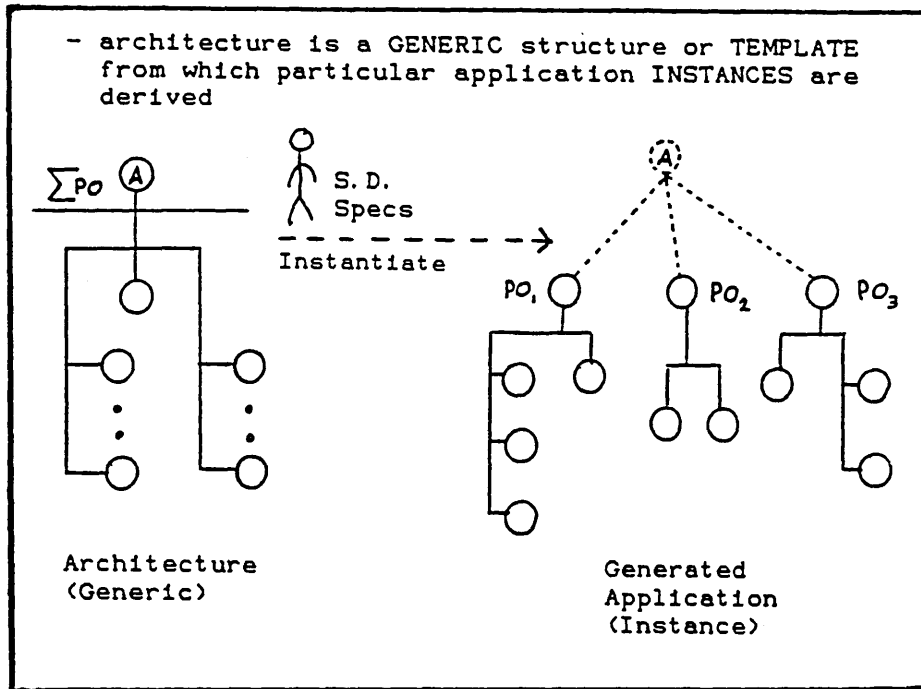


Figure 2.4 - Database Applications as Instantiations

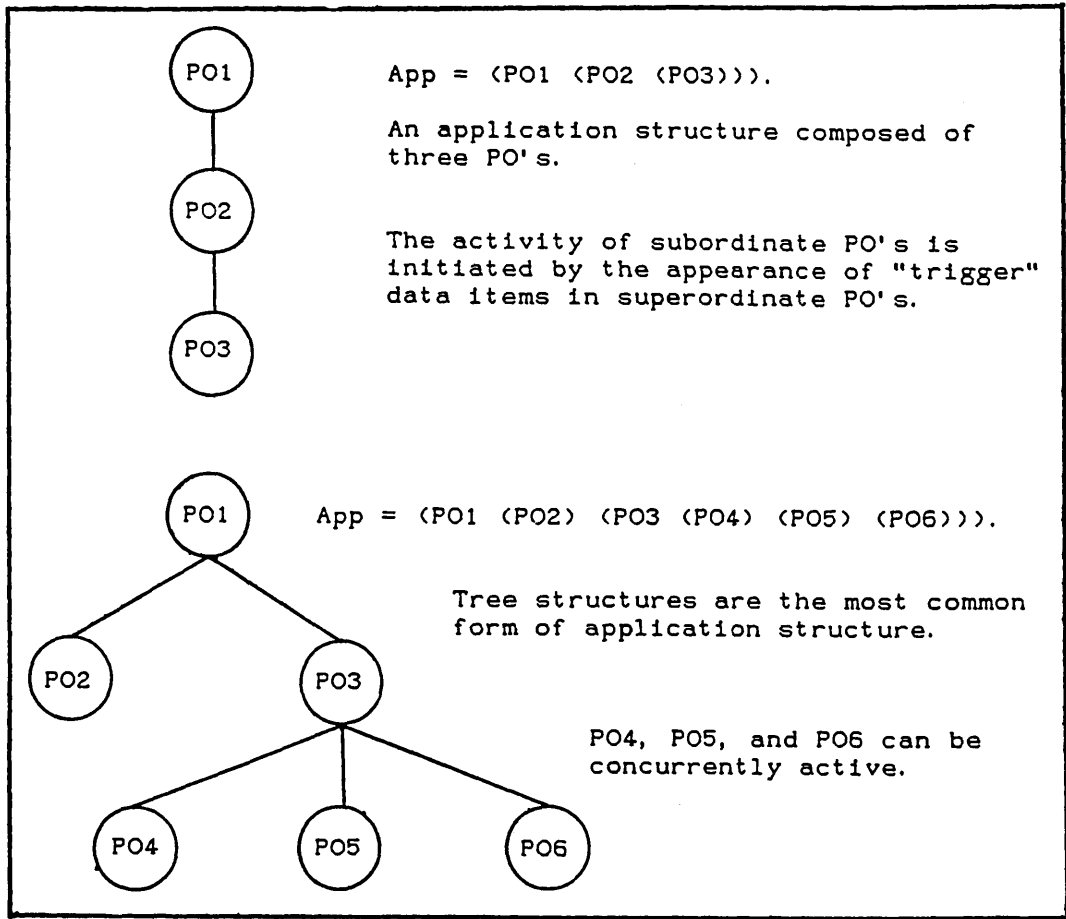


Figure 2.5 - Examples of Application Structures

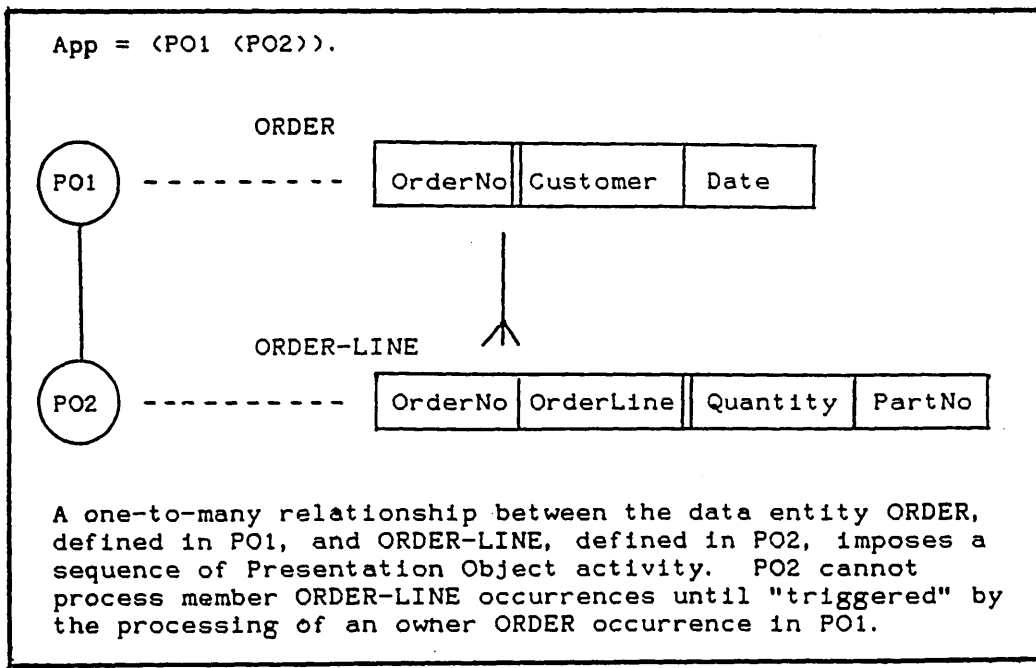


Figure 2.6 - Data Dependencies and Object Activity

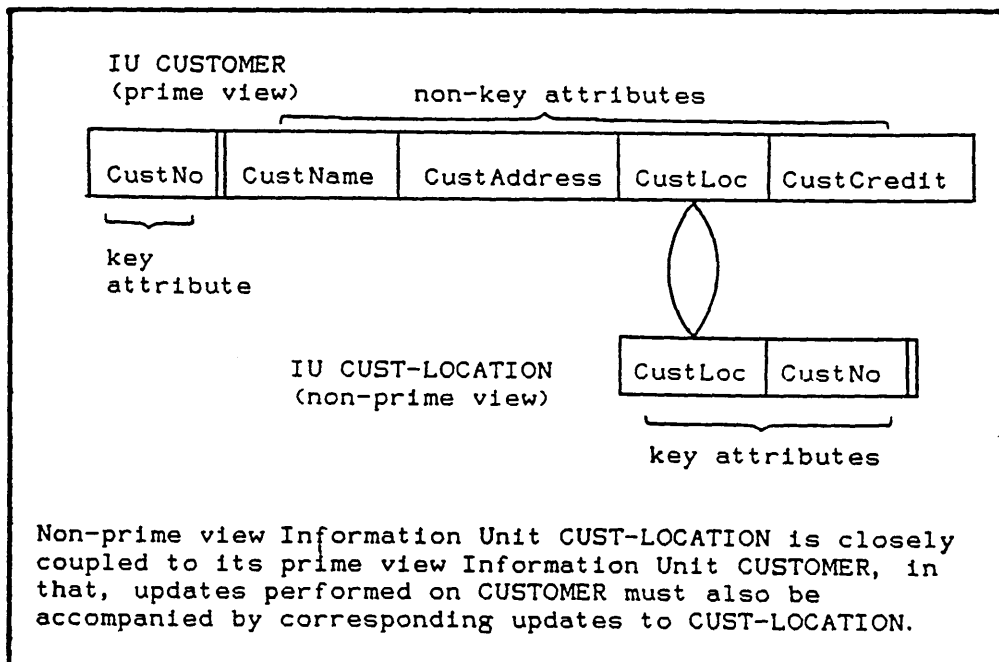


Figure 2.7 - Prime View and Non-prime View Information Units

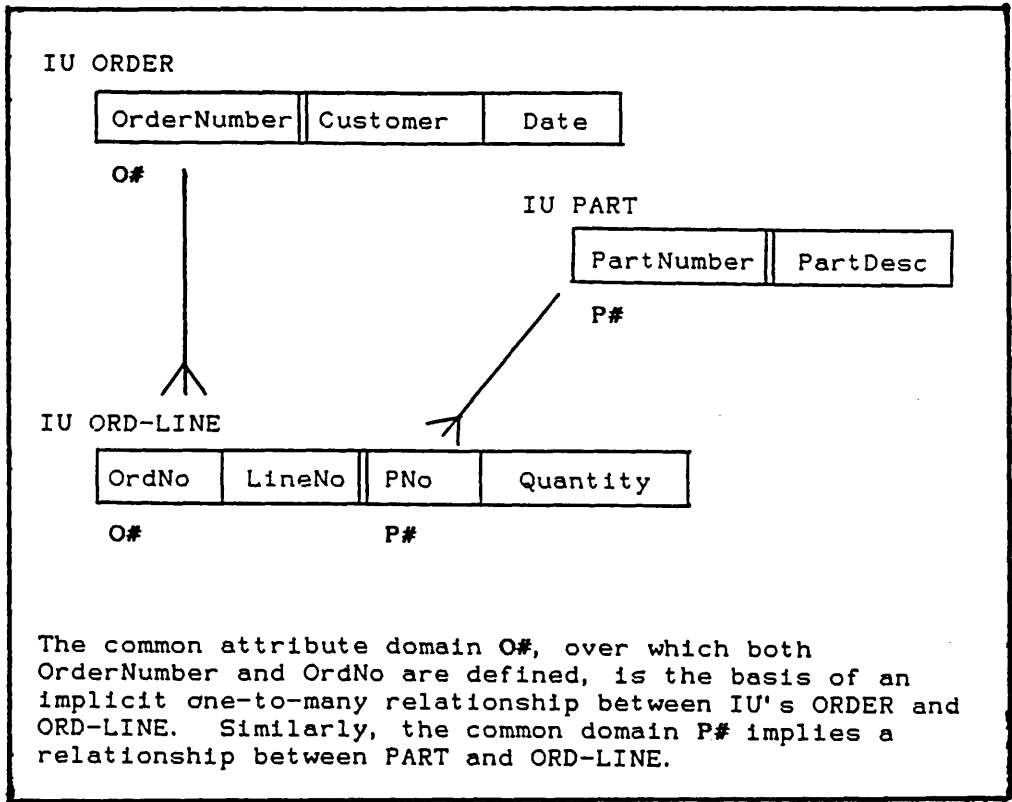


Figure 2.8 - Domains and Relationships

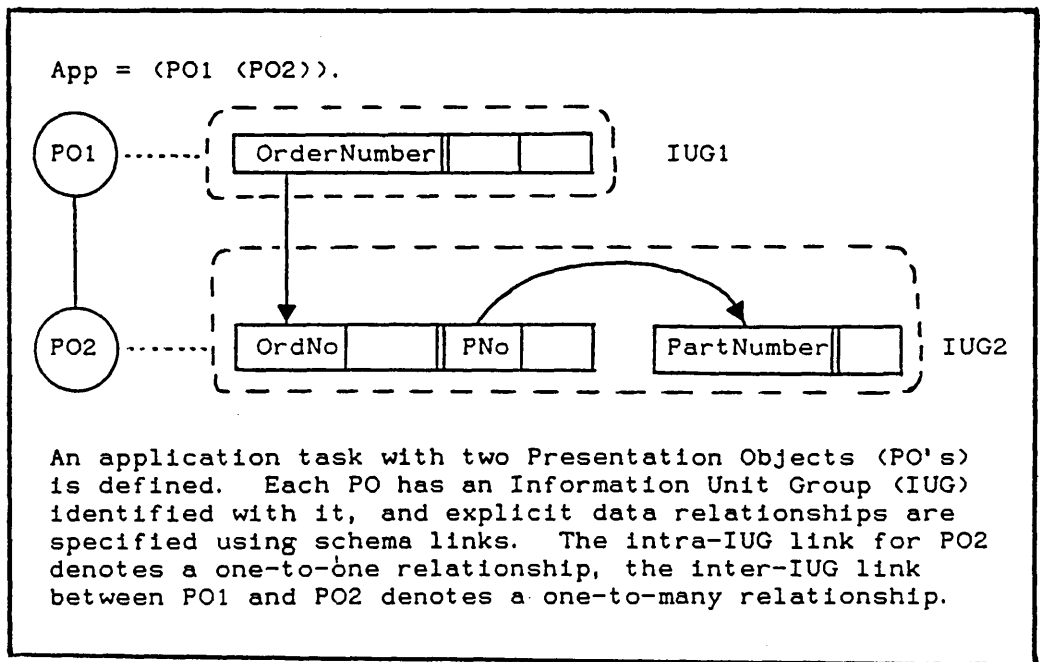


Figure 2.9 - Information Unit Groups and Schema Links

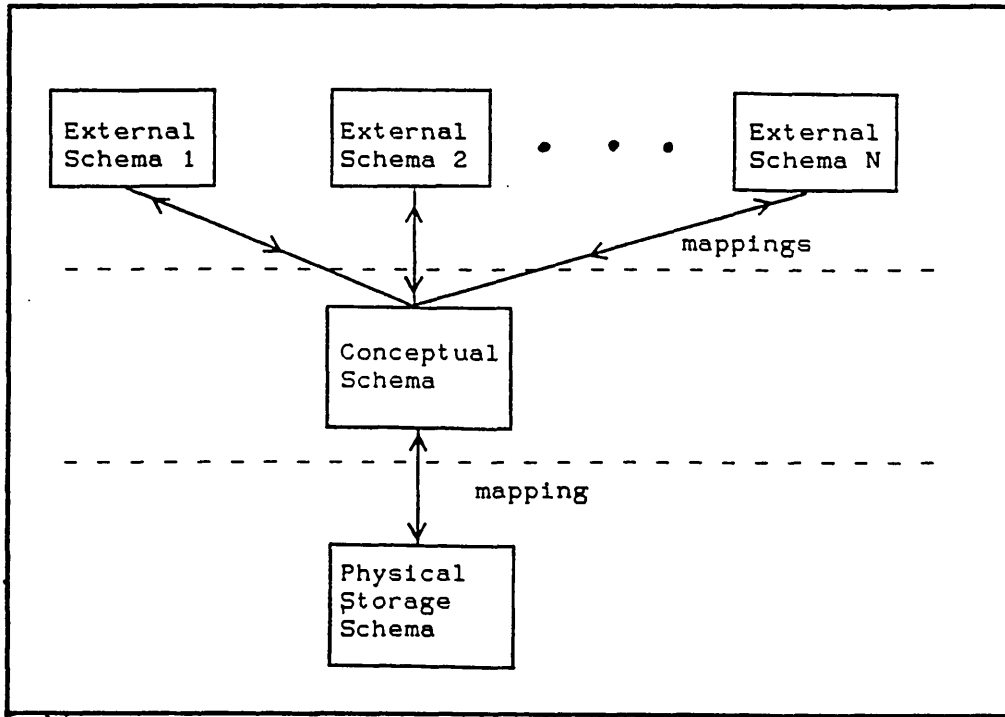


Figure 2.10 - ANSI/SPARC DBMS Model

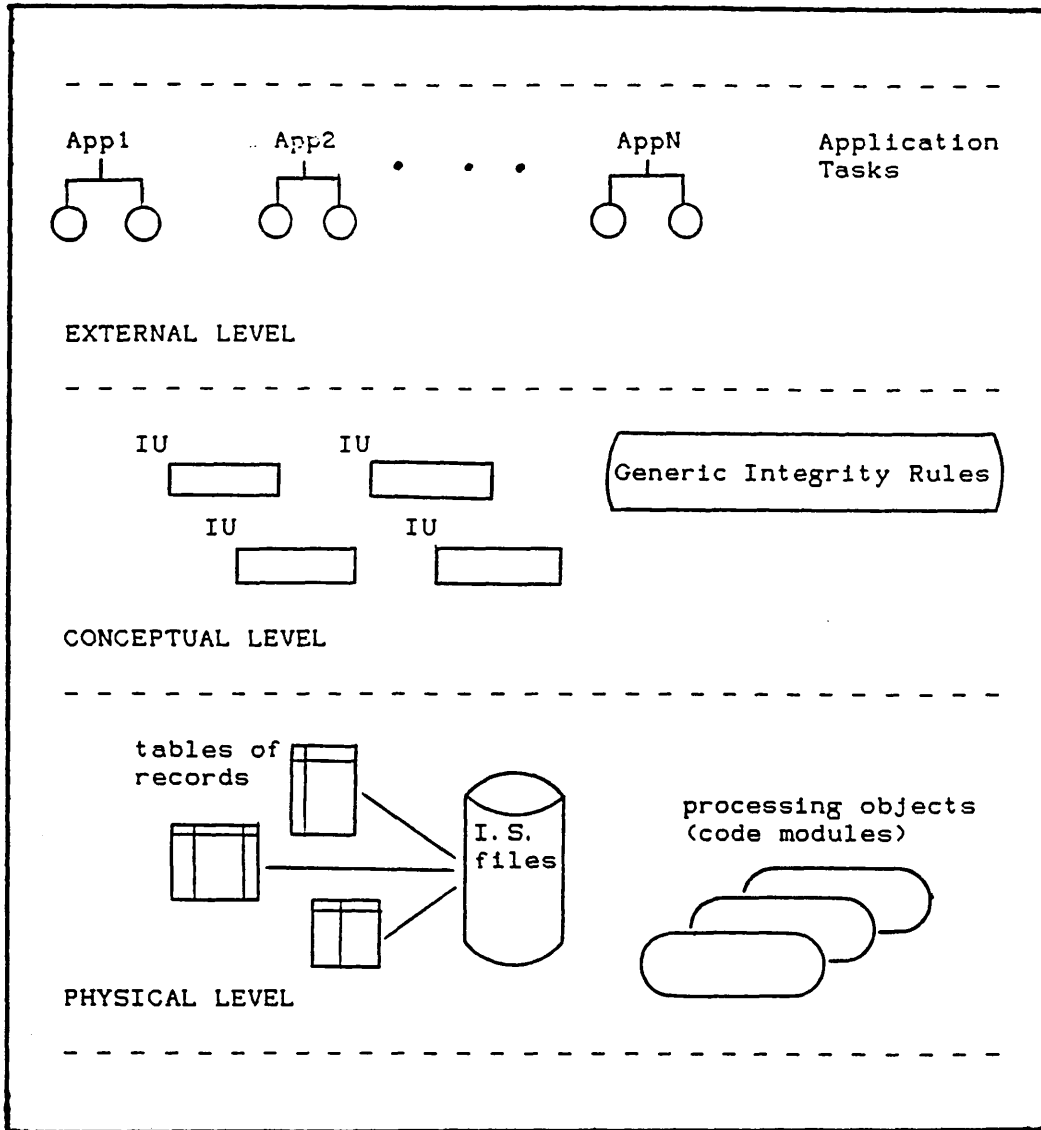


Figure 2.11 - Levels of DB4GL Data Description

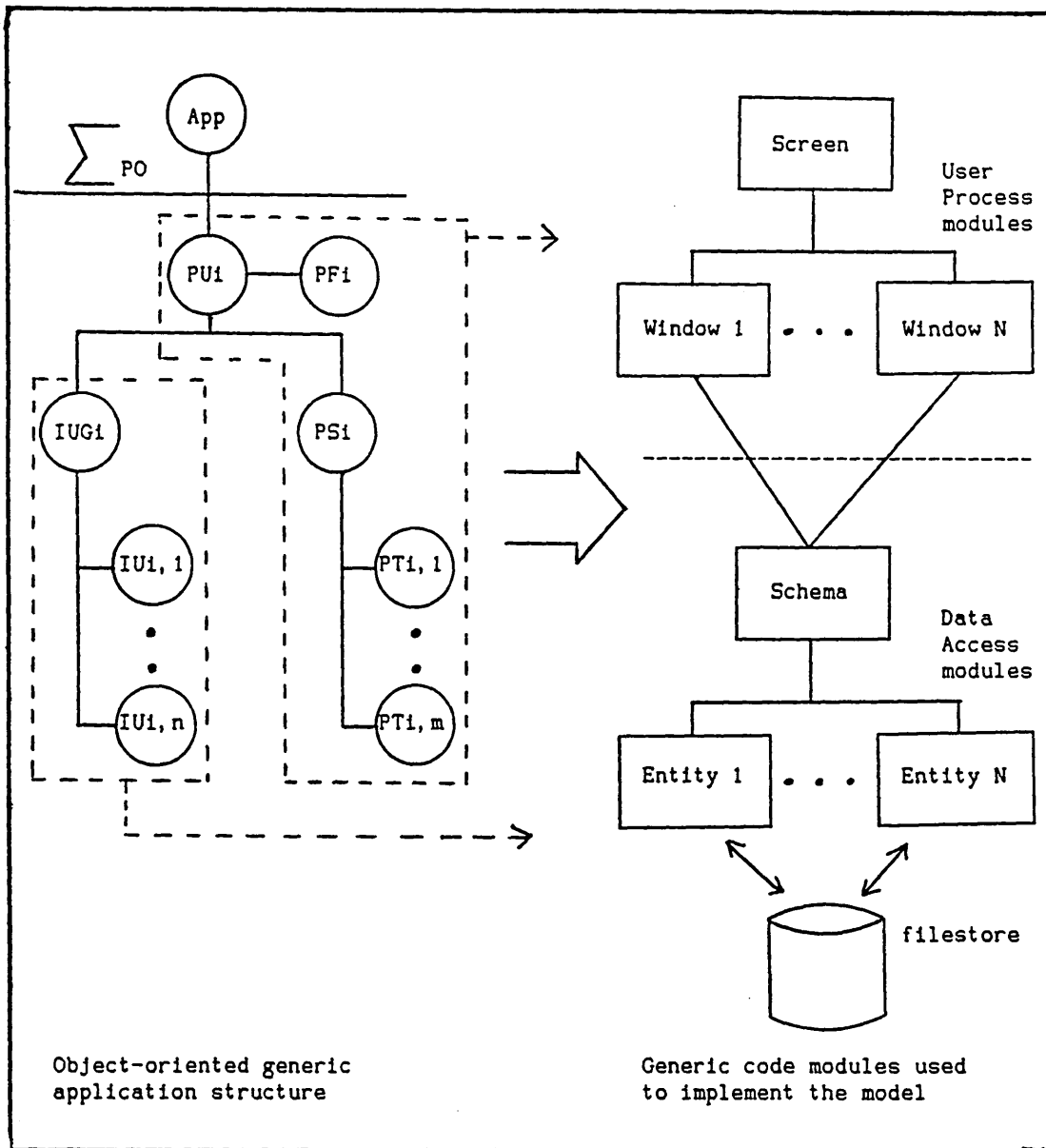


Figure 2.12 - Implementation of the Application Model

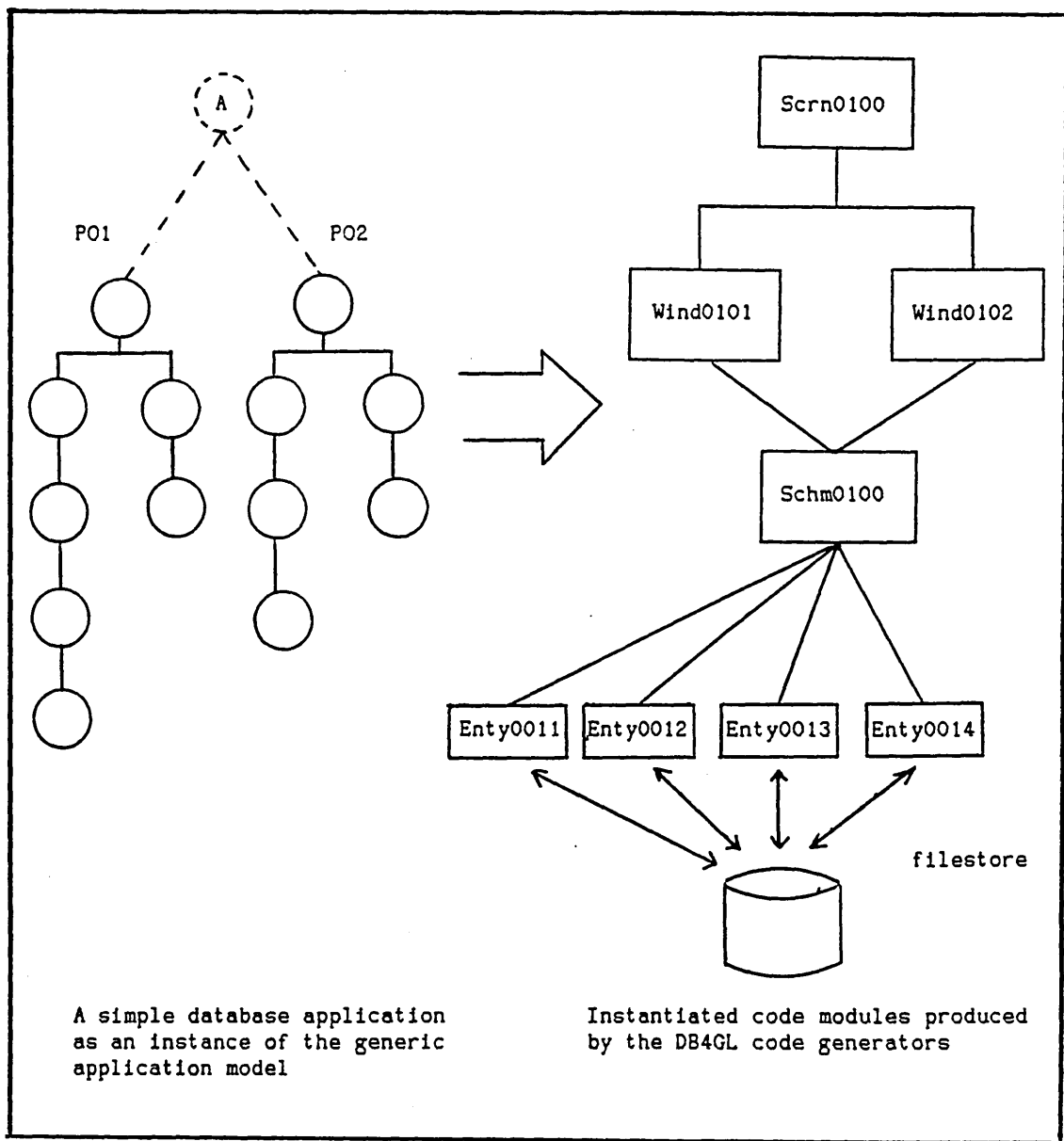


Figure 2.13 - Generated Application Instances

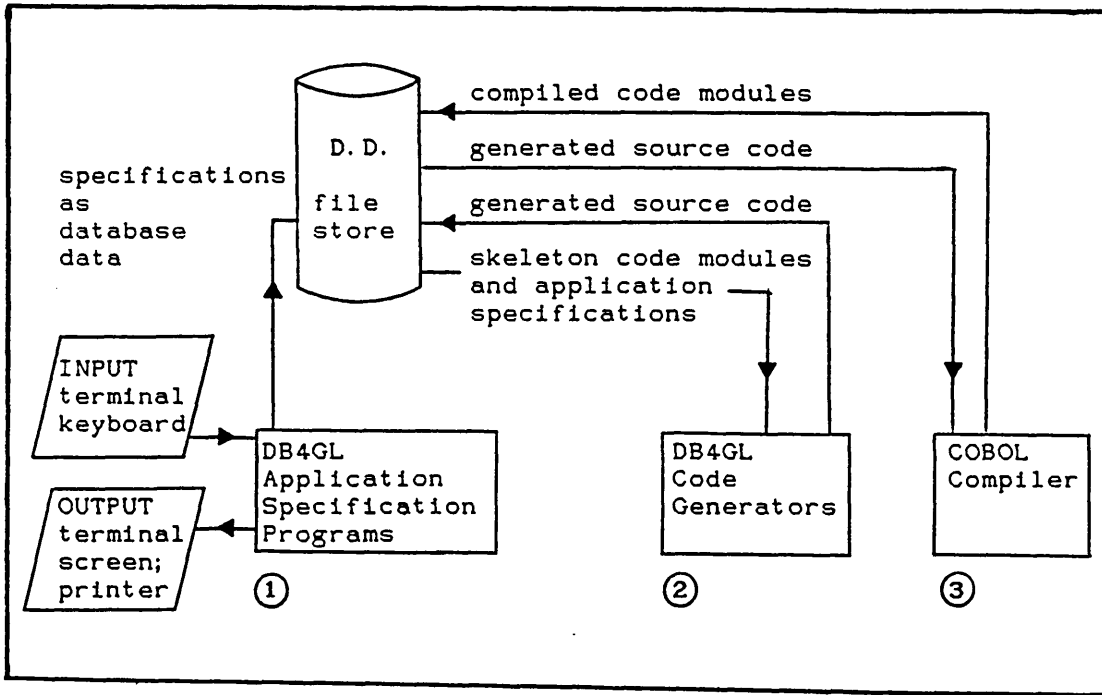


Figure 2.14 - DB4GL Code Generation

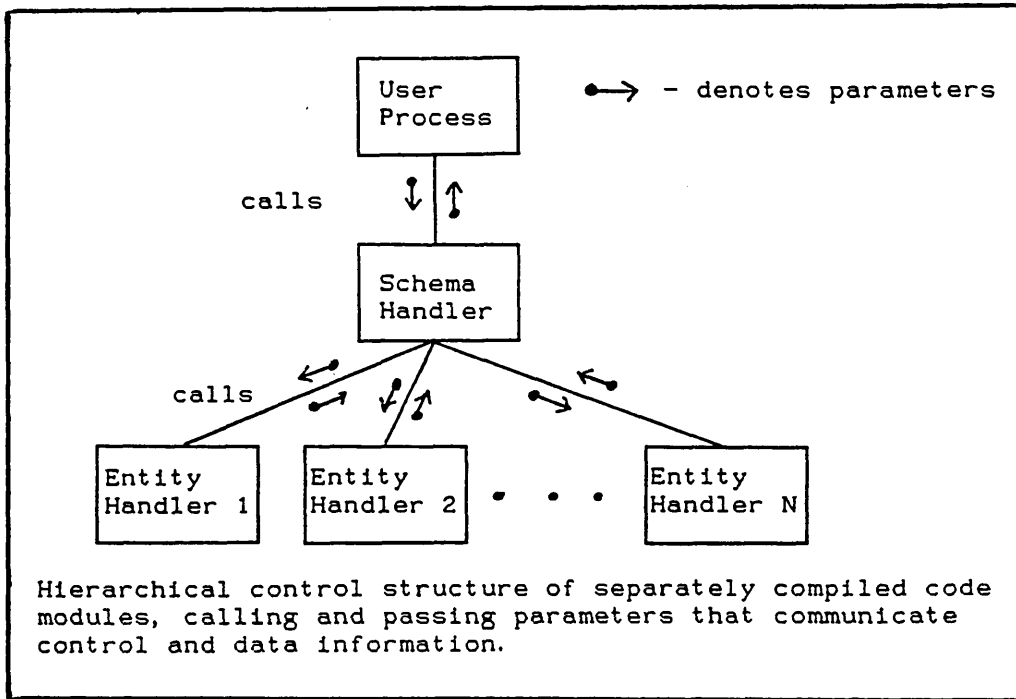


Figure 2.15 - Modular Structure of Implemented Applications

Object-Oriented Database Approaches

3.1 A Classification of Object-Oriented Approaches

The "object-oriented" literature is vast. However, a useful collection of papers giving an introduction to this area can be found in Proc. of ECOOP88 [Gjessing88]. The terminology in the area of object-oriented programming in general, and object-oriented databases in particular, tends to be inconsistent and somewhat confusing. There is some disagreement about what an object-oriented database is, authors in this field each have their own opinions. The position taken in this thesis, is not to give the definition of "an object-oriented database"; but to first identify a number of object-oriented database approaches, and then use this as a framework in which to explain the influence and relevance of object-oriented concepts to DB4GL

An object-oriented database approach is a theoretical or practical approach to the design and implementation of database systems that is influenced by the object-oriented perspective [Lindsjorn88]. This perspective is derived from object-oriented programming [Cox84] [Cook86] [Stefik84], which is characterised by the following features:

- type/instance differentiation;
- encapsulation of code and data within objects;
- inheritance of object properties via a generalisation/specialisation hierarchy;
- inter-object interaction via message passing.

It is possible to identify a number of database approaches influenced by the object-oriented perspective. A (non-exhaustive) list of the main object-oriented database approaches can be made:-

Approach I

The enhancement of data modelling languages and DBMS's with object-oriented concepts and facilities (such as generalisation and inheritance hierarchies), for example, Iris [Fishman87], OODM [Zhao88], Cactis [Hudson89]; this approach can be partly seen as an extension of semantic database models. The main aims of this approach are: to capture more application domain semantics in the data model; to represent complex data objects; and support multi-media data. Usual accompaniments to this approach are: sophisticated user

interfaces; support of functionally defined (derived) data; and concurrent/parallel implementations with message-driven computation.

Approach II

The use of an object-oriented development approach [Booch86] [Korson90] (Figure 3.1) to the specification and implementation of database systems, for example [Baroody81] [Neuhold86] [Sernadas87]. This has similarities with the Abstract Data Type (ADT) approach to software engineering [Kerridge89], and, to a lesser extent, with the modelling approach of JSD [Jackson83] [Hull89]. In this approach (II), the principal object-oriented concerns are those of: encapsulation/abstraction; system decomposition based on the object concept; the use of object structures as models of the real world; and issues of concurrency and communication. A recent development in this area is the amalgamation of object-oriented design with structured design methodologies, for example [Wasserman90].

Approach III

The use of an object-oriented programming language (OOPL) to implement database systems. This could be an object-oriented language such as Smalltalk [Xerox81] or C++ [Stroustrup86]. Alternatively, it could be an OOPL specifically tailored for programming knowledge-based/database systems; for example OOPS+ [Laenens88], in which database concepts are integrated with knowledge representation techniques such as demons and rule based inference. Note that - approach (III) is different, because approaches (I) and (II) do not necessarily involve an object-oriented programming language; also, the implemented database system might not be recognized as an object-oriented DBMS of the type described in approach (I).

Approach IV

Interfacing a database system to an object-oriented programming environment, whereby a conventional relational DBMS maintains the persistent data, and an interface component constructs from this data, complex objects required by the object-oriented environment (OOE) (Figure 3.2). An example of this approach can be found in [Wiederhold86] which emphasises the similarity between objects in an OOE and the view concept in relational databases. A single complex

object when stored in a relational database is decomposed into a number of tuples which are distributed across many separate relations. When this complex object is retrieved from the relational database it has to be reconstructed from the many separate tuples; this is effected by the application of relational operations to the appropriate relations. Views in a relational database can be defined by the application of relational operations to a number of base relations. Thus, there is a similarity between the storage and retrieval (decomposition and reconstruction) of complex objects and the maintenance of views in relational database.

3.2 Object-Oriented Approaches and DB4GL

Such a crude taxonomy of ideal-typical object-oriented database approaches inevitably oversimplifies the situation. Any particular example of an actual object-oriented database project will normally contain elements of more than one approach. In the DB4GL project, there is a mixture of principally approaches (I) and (II). The vocabulary of the object-oriented paradigm, in particular, the terminology of "mainstream" OODB research (approach I) is used in the description of the DB4GL application model. This is not because DB4GL was originally intended to be an example of such an OODB system, in fact, the original motivation and direction of DB4GL research was in the area of application generation and prototyping system development. In the search for a unifying formal structure for database application specification and generation the DB4GL application model has been developed, and this model has acquired several features normally associated with the object-oriented paradigm.

A key feature of the application model is the grouping together of database data and processing, along with integrity rules and user interface, and their incorporation into entities known as Presentation Objects (PO's). PO's not only unite these different facets of a database application, but also effect a partitioning of the application's database data. The interaction between the loosely coupled independent PO's that constitute a DB4GL database application is primarily based on data-flow realised through message passing. As data attributes within PO's become active (ie derived data is generated, or persistent data is updated) messages containing control and data information are communicated between the PO's. Thus, the course of computation within a DB4GL application is essentially message driven.

Central to the operation of DB4GL as an application generator is the self-describing DB4GL data dictionary. The data dictionary contains a description of the application model, and is also used to store the specifications of particular database applications. These specifications are in terms of Presentation Objects, combining data with processing and integrity rules. For this reason, the data dictionary can be described as an object dictionary. Furthermore, the relationship between the application model and the generated database applications is one of generic type to particular instances; that is, the generated database applications can be considered as instantiations of the DB4GL application model.

The object-oriented development approach (II) has been another source of object-oriented influence upon the DB4GL project, particularly with respect to the software engineering task involved in the construction of a parallel version of DB4GL. A specific object-oriented design methodology has not been adopted in either the original DB4GL implementation or in the P-DB4GL implementation. However, the principal concerns of this approach (II) are addressed in the DB4GL implementations; less so in the original implementation, but more obviously in the parallel P-DB4GL implementation. In the original DB4GL implementation, the design translation from specification objects to executable code modules endeavoured to produce well factored modular program designs with clearly defined module interfaces. The message driven computation inherent in the application model was retained, but the implementation in a sequential programming language necessitated a data flow inversion, converting the inter-object message passing into parameter passing at sub-module calls. The object-oriented development concern with the use of object structures as models of the real world is reflected in the use of Presentation Objects to define applications. For example: an ORDER PO corresponds to an item of concern in the application domain, and in common with the real world Order item, the ORDER PO encapsulates the many different properties (such as data attributes, processing/methods, life history/update dependencies, presentation/interface) of this Order item.

The influence of the object-oriented development approach (II) is far more significant in the P-DB4GL project. The principal design approach has involved a relatively direct translation from specification objects to implementation processes. In order to construct the prototype P-DB4GL

implementation this design approach has been somewhat compromised, and the translation made less clear, the reasons for this compromise are given in Chapter 5 section 5.4. The objective of this translation was to retain the parallelism inherent in the specification of an application as a set of independent, concurrently executable, Presentation Objects, and transform it into an explicitly concurrent process based program suitable for execution on parallel transputer hardware. The process based implementation language (Occam) provides good encapsulation of data and code. In the course of P-DB4GL development certain design concerns have been examined in great detail. For example, the granularity of concurrency: Occam permits arbitrary levels of concurrent decomposition, whereas some other concurrent design methodologies (JSD) only permit a single level of concurrent description. The choice of suitable concurrent units has been very important in P-DB4GL designs, especially when considering how the concurrent programs are to be distributed. Also, the details of the communication mechanisms used to implement inter-object message passing (such as synchronization, buffering, layered protocols, message formats) have been examined to a greater degree of detail than might be encountered in some design methodologies.

There is no evidence of the third object-oriented approach (III) within the DB4GL project. The DB4GL systems have not been implemented in an object-oriented language or environment - the original version was implemented in COBOL, whilst the parallel version used Occam. Although the object-oriented development approach (II) has been influential, the process based concurrent Occam language has been used in preference to an object-oriented language. This is in order to extract the maximum possible performance from the transputer based parallel hardware. During the development time of the P-DB4GL project, no fully functional example of an object-oriented programming environment has been available on transputers. It is doubtful whether such an environment would permit full advantage of the transputers' parallel performance. The processing overheads and types of communication mechanism provided by a general object-oriented environment may not be suitable for the specific requirements of the DB4GL system.

The remaining object-oriented approach, interfacing an object-oriented environment (OOE) to a relational DBMS (approach IV), is not present in the DB4GL project. In fact, this approach is not at all appropriate to DB4GL (original and parallel versions), for the following reasons. First, DB4GL does

store persistent data in a tabular form, but DB4GL is not strictly relational. Secondly, DB4GL is not a conventional DBMS, it is an application generation environment in which database applications are produced in their entirety, there is no distinction between separate application programs and DBMS functionality. Thirdly, the Presentation Objects (PO's) of a DB4GL application are of a very different scale and purpose to the individual complex objects manipulated in an OOE. The PO's of a DB4GL application serve to partition the underlying persistent data within an application, and define the application's processing and interface activity upon that partition. There is no notion of fragmentation and reconstruction of individual complex object instances associated with the storage and retrieval activities present in approach (IV).

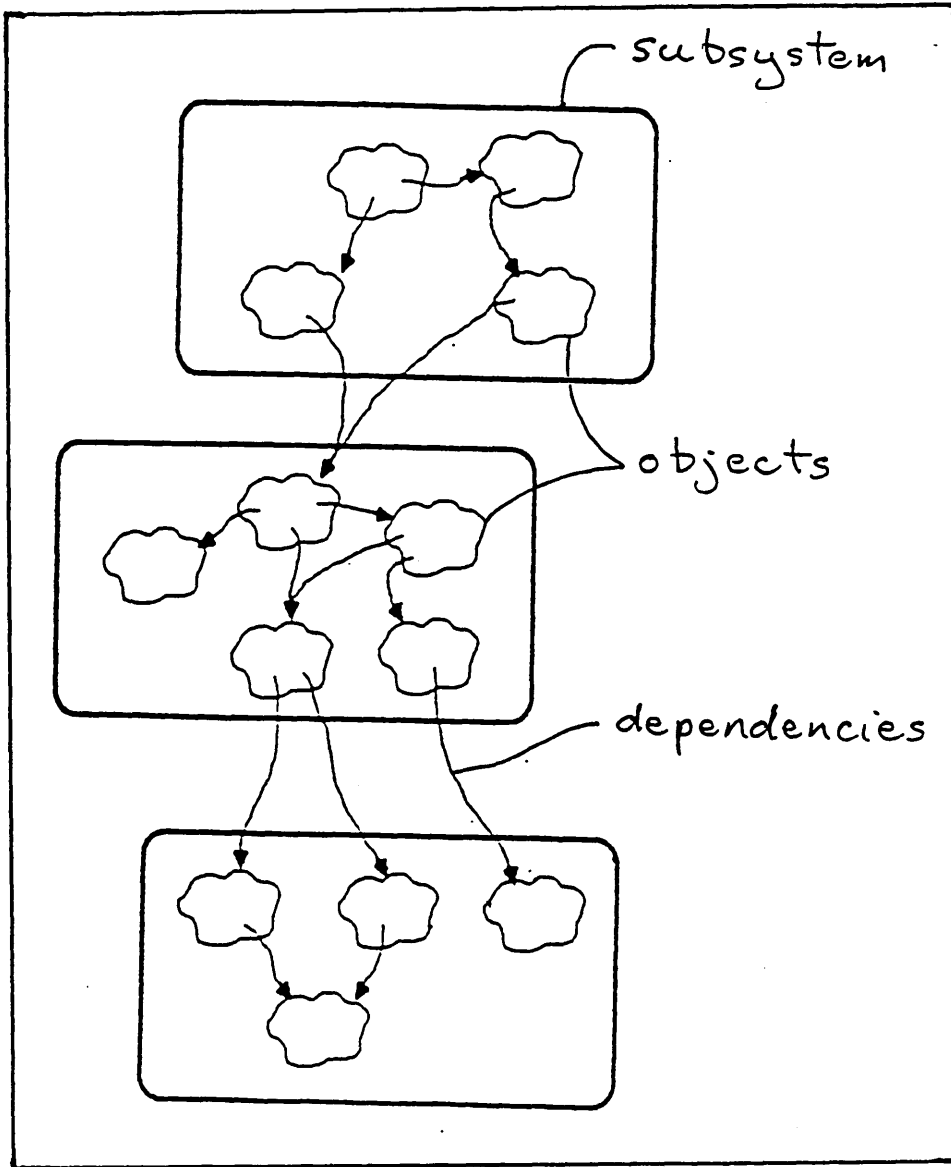


Figure 3.1 - An Object-Oriented Design

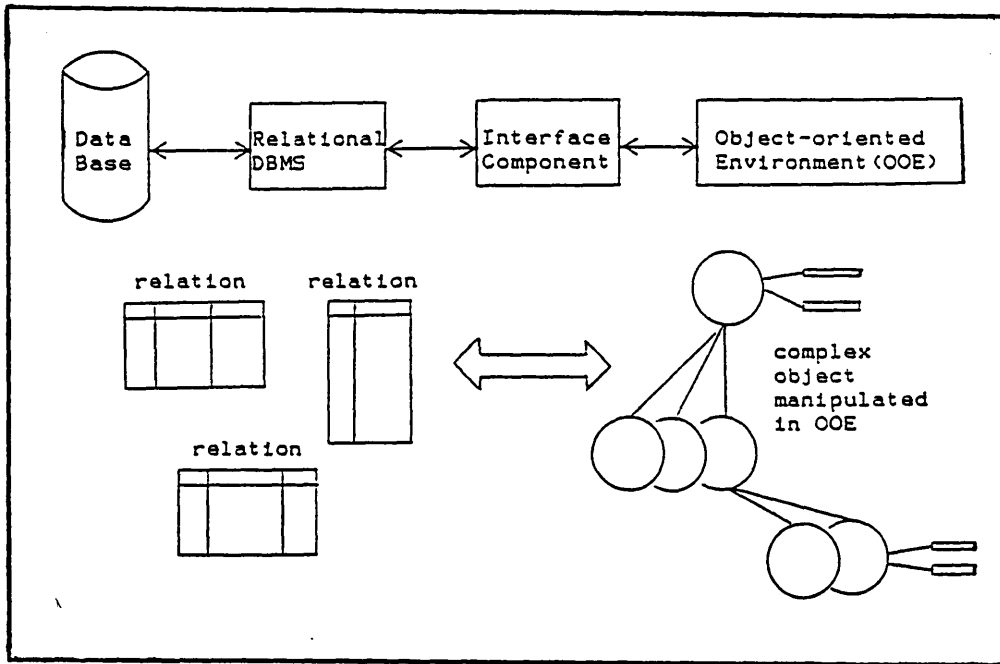


Figure 3.2 - DBMS/Object-Oriented Environment Interface

Chapter 4

Transputers, Occam, and Databases

4.1 Transputers

Transputers are a family of programmable VLSI devices produced by Inmos Ltd [Inmos89b]. Transputers are principally designed for the construction of parallel processing architectures based on local memory and point-to-point serial communication [Inmos89a]. The family of devices includes: microprocessors; dedicated peripheral controllers; digital signal processors; and communications devices. A typical transputer 32 bit microprocessor (eg T414-20) contains in a single integrated circuit: a CPU; internal memory (2K on-chip RAM); an external memory interface; four high-speed (20 Mbit/s) bidirectional serial links, and is capable of 20 MIPS (peak) instruction rate (Figure 4.1). Transputer microprocessors have a low level microcoded scheduler which enables any number of processes to execute together on a single processor, with each process sharing processor time.

Using transputers, parallel systems are built as networks of microprocessors and peripheral controllers connected by the devices' point-to-point serial links. Various network topologies can be constructed, for example, rings, trees, regular arrays, and hypercubes. Because there is no shared memory or communication bus connecting the processors, the communication bandwidth of a transputer based system grows in proportion to the number of processors present. Thus, transputer based parallel architectures tend to be very scalable, it is usually just as simple to construct a ten processor network as a thousand processor network. However, not all parallel algorithms can benefit from such large numbers of processors, and though the parallel hardware may be readily scalable, many parallel algorithms are not.

4.2 Occam

Occam [Inmos88b] is a high level programming language designed for concurrent programming and derived from Hoare's CSP language [Hoare78]. Occam is based on a process model of computing, and uses unbuffered unidirectional point-to-point channels for inter-process communication, with the communication between connected processes synchronizing otherwise independent processes (Figure 4.2). The transputer and Occam

were designed together, every transputer implements the Occam concepts of concurrency and communication, and the transputer instruction set contains instructions for the optimal implementation of Occam [Inmos88d].

When a transputer network is programmed in Occam, configuration information has to be supplied to an Occam program. This configuration information places particular processes at numbered processors in the network, additionally, the Occam channels connecting processes have to be mapped onto the point-to-point serial links connecting the transputers. Configuration information is static, and fixed when a compiled program is linked. All the code intended to run on a processor must be explicitly placed on it during configuration. Occam programs do not allow one process to remotely invoke another process placed on a separate processor. Operating systems, providing facilities such as dynamic load balancing of networks, message routing, program scheduling, and filestore management, are now becoming available for transputers, for example Helios [Perihelion89] [Grimsdale89] and Mercury [Oakley89]. However, it is more usual for transputer applications to run without operating system support; applications are typically developed on a host computer (such as a PC), then the compiled, linked and configured programs are loaded onto the target transputer network.

4.3 Parallelism and Databases

Parallel architectures are being applied to database applications in a number of ways. One type of approach is to transfer some of the database processing normally performed in the CPU into the storage devices. Using specialized hardware, these associative storage devices can perform database operations equivalent to relational selections and projections, thus relieving the CPU of such computationally intensive tasks. These devices are essentially single instruction multiple data (SIMD) architectures, and generally only implement specialized search functions of a DBMS, most of the DBMS code executes on a separate host machine. Some of these associative storage devices perform "logic per track" processing [Parker71], for example RAP [Ozkarahan75] [Ozkarahan77], CASSM [Su75a] [Su75b], and RARES [Lin76]. Other devices such as CAFS [Babb79] do not have a processor per track, but process a number of multiplexed disk channels.

A different type of parallel approach is to execute all of the DBMS on a multiple instruction multiple data (MIMD) architecture constructed from a collection of storage devices, memory modules, and processing units interconnected in some fashion, for example DIRECT [DeWitt79]. A classification and evaluation of these multiprocessor architectures is provided in [Stonebraker86], in which three classes of architecture are identified:

- shared memory (SM);
- shared disk (SD);
- shared nothing (SN).

An advantage of SM architectures, for example XPRS [Stonebraker88], is that generally fewer modifications to algorithms are required when porting DMBS software from a non-parallel machine. However, SM architectures suffer from poor scalability, the common central memory becomes a highly contended resource as the number of processors is increased. In SD architectures, such as Amoeba [Shoens85], VAX DBMS [kronenberg86] [Rengarajan89], and DCS [Sekino84], each processor has its own private memory, and memory contention is not a problem. But as SD architectures are scaled up to larger sizes, the shared disc channels become a point of high contention with consequent performance limitation.

In SN architectures, neither memory nor discs are shared amongst processors, each processor has its own private memory and local disc; consequently contention is not a problem as the architecture scales up in size. Examples of SN architectures include: Terradata DBC [Neches86] [Terradata88]; Bubba [Alexander88] [Boral88] [Copeland88]; and Gamma [DeWitt88] [Schneider88]. However, SN architectures can be prone to message handling delays. In order to reduce the number of inter-processor messages, and keep message distances to a minimum, processing of disc resident data should be as kept as local as possible, and great care is needed in database design. Stonebraker's opinion (expressed in [Stonebraker86]) is that most transaction-oriented database applications are amenable to the parallel design tuning needed in SN architectures, and that SN architectures offer the best opportunity for parallel speed-up and scale-up.

Transputers offer a number of advantages as the basis of parallel hardware for database applications. Each transputer contains processor, memory, and communications facilities in a single package, and large distributed memory

architectures can be constructed in which the communications bandwidth grows proportionately to the number of processors. Special peripheral interface transputers, such as the M212, permit disc storage to be easily integrated into a distributed transputer based architecture, making SN type database architectures simple to construct. Furthermore, transputers offer a relatively inexpensive entry path into parallel architectures, they are available as "off the shelf" packages conforming to published standards. Transputer based systems are highly extensible, and provide significant advantages, both to researchers and commercial system designers, over parallel database architectures based on specialized custom built (and generally expensive) hardware.

Transputer based parallel architectures have already been used in database applications in a number of ways. One approach, for example [Stringer89], is to download an entire database into a transputer network, all the database data can then be searched in parallel, with each processor searching the database partition held in its own local memory space. Obviously, such an approach is only possible if the database is small enough to be temporarily held in main memory. A 32 bit transputer has a maximum address space of 4 GigaBytes, and the total available memory in a network can grow proportionately with the number of processors. However, there are a number of problems associated with such "main memory" database systems, both on transputers and on other parallel hardware. The time needed to transfer a large database from disc to memory before processing can commence, can be quite considerable. This would tend to restrict such an approach to applications with (relatively) small databases which are either static or else occasionally updated off-line. If updates are performed on-line to an "in memory" database, some provision for saving the updated data on non-volatile (magnetic) secondary store must be made for recovery purposes. If a database is small enough, it may be possible to hold it entirely in non-volatile electronic store, however the cost of such non-volatile memory makes such a solution infeasible for most databases.

For database applications with very large amounts of updatable data, which for reasons of size and/or cost require magnetic secondary (disc) storage, different transputer based approaches are being tried. One is DRAT [Kerridge87], a design for a relational database machine using dynamically reconfigurable networks of transputers. The DRAT design incorporates several transputer controlled hard disc drives and C004 transputer crossbar

switches: relations are read from disc and the tuples are "piped" through a network of transputers in which relational operations such as joins are performed in parallel. Another approach is LSDM [Rishe89] [Li90], for a database system based on a Semantic Binary Model; this proposes a hypercube architecture of many transputers each with its own small local disc, in which as much processing as possible is performed locally without having to move data to other nodes.

The main advantages of transputer based architectures for database systems can be summarised as:

- scalability of communications and memory bandwidths in proportion to increasing numbers of processors;
- proximity of processing and data afforded by tight coupling between processors and discs, made possible by the processors' on-chip serial links and peripheral controllers using these high speed links;
- high disc I/O bandwidth, made possible by using a large number of discs, these can be small cheap Winchester discs.

All the above features have been utilised in the design of Parallel-DB4GL (P-DB4GL), but the P-DB4GL system has not been designed for a specific network topology. Also, the P-DB4GL project has been restricted in considerations of physical storage details to the level of the disc interface (ST506, SCSI), that is simply to reading and writing of blocks of records from disc sectors. Although it has been assumed that small Winchester discs would predominantly be used, the possibility of incorporating "silicon discs", particularly for optimising access to frequently accessed tables, has not been excluded.

4.4 Project Resources

The Department of Computer Studies at Sheffield City Polytechnic has made a number of transputer resources available to the P-DB4GL project. This includes: two development environments, the Transputer Development Environment (TDS) [Inmos88c] and the Occam 2 Toolset [Inmos89d]; several boards containing transputers (B004, B006, B008 boards and TRAMs [Inmos89c]); additional language compilers (C and FORTRAN); and a transputer controlled Winchester hard disk board. These resources have been extensively used in both the development of the prototype P-DB4GL system and the partial implementation of the fully functional P-DB4GL.

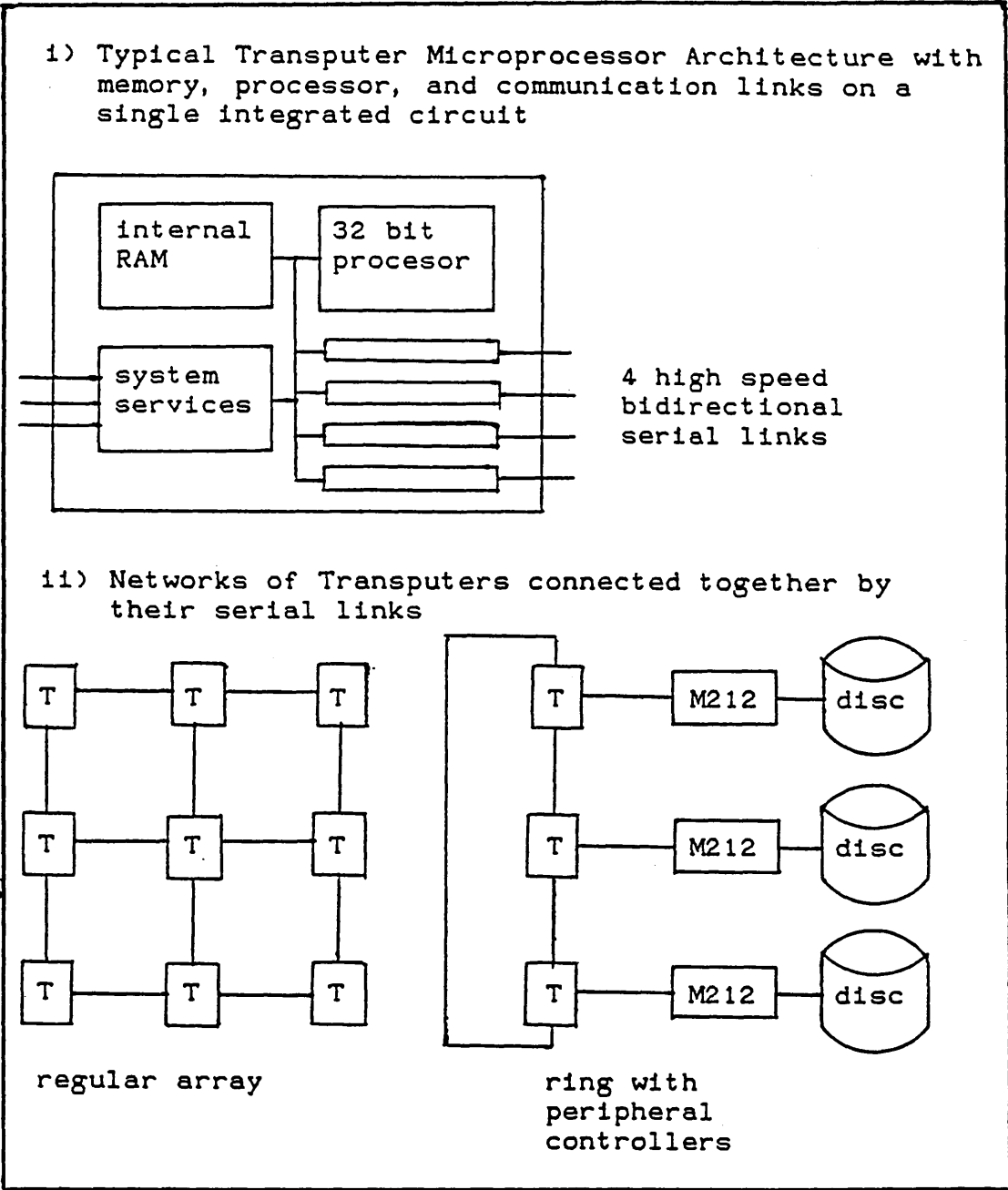


Figure 4.1 - Transputer Architecture

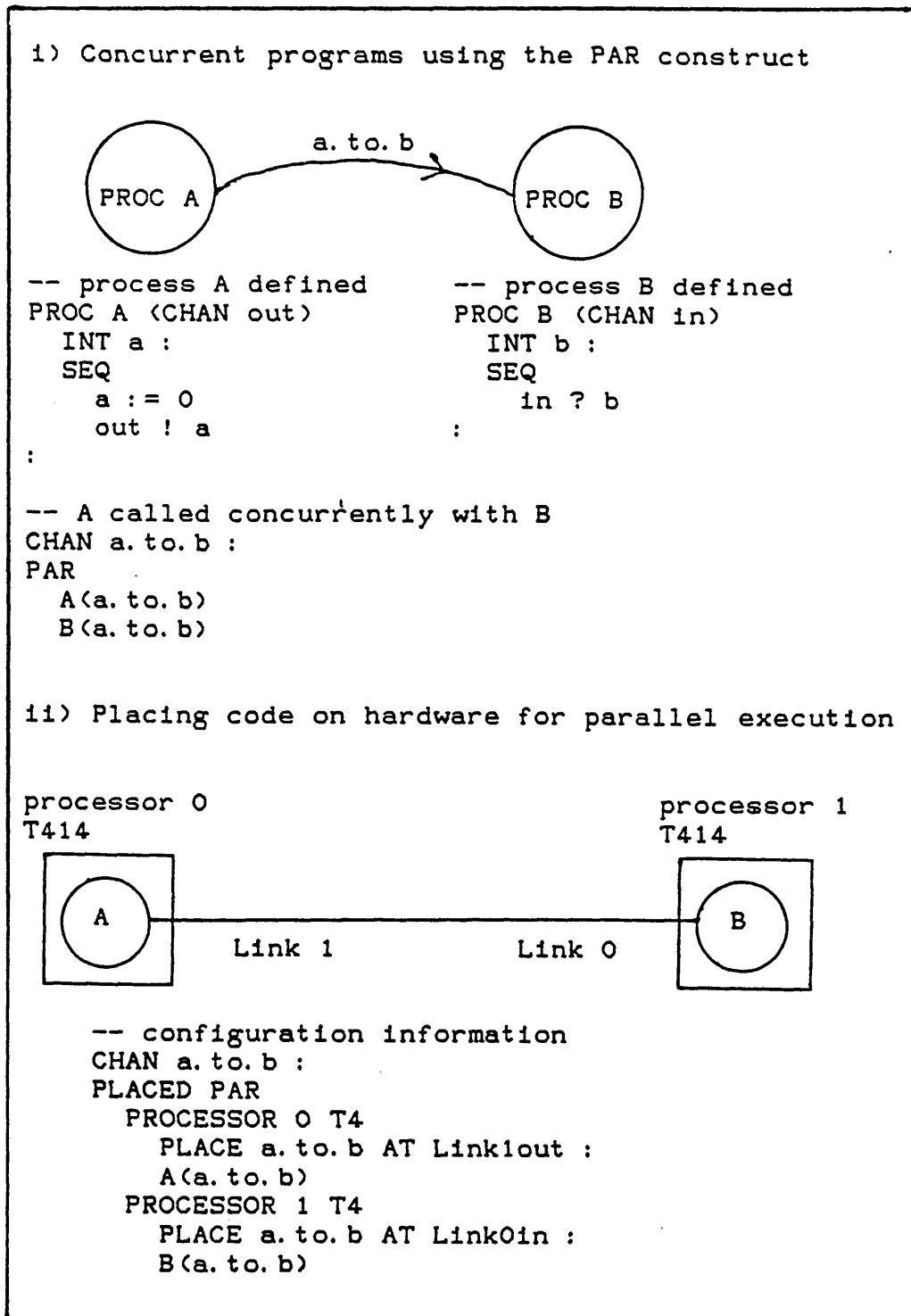


Figure 4.2 - The Occam Language

Chapter 5

Developing the Prototype Parallel-DB4GL

5.1 Project Objectives

The Parallel-DB4GL (P-DB4GL) project is an extension of the DB4GL research programme. It centres on an investigation into ways of enhancing the performance of the DB4GL generated database applications by using parallel processing hardware. Previous DB4GL research [Hird89] suggested that performance gains could be obtained from running DB4GL applications on transputer based parallel architectures [Inmos88a]. The principal aims of the P-DB4GL project are:

- determine the exact amount of parallelism inherent in DB4GL database applications;
- develop suitable hardware and software architectures for execution of the applications on parallel processing hardware;
- test the feasibility of using transputers as the basis of the parallel processing hardware.

The parallel hardware should be transparent to the end users of the generated applications. The P-DB4GL applications should look and behave exactly like the sequential DB4GL, the only difference being improved execution times. The parallel hardware - transputer microprocessors, disc controllers, and disc drives - is carried on PC expansion boards, mounted either inside the host PC or externally in a separate rack. All of the parallel DB4GL database application code, and the DB4GL system tools (such as the code generators), executes on the transputers, the host PC is used simply as a terminal interface for the application user (Figure 5.1). The DB4GL development cycle remains unchanged: the database application is generated from the specifications entered to the data dictionary by the system designer, but the generation tools are redesigned to produce a concurrent program for parallel execution on the transputer network.

5.2 Design Methodology

The DB4GL application model or "architecture of a database application" contains a high degree of inherent parallelism. The constituent Presentation Objects (PO's) of an application task are independent, though related, processing objects, and in principle are capable of concurrent

operation. The interaction between PO's consists of communicating data and control messages. An application task can be viewed as a network of PO's, with data/control messages flowing between related PO's. The PO's are related by the data dependencies between the data entities (Information Units) processed by each PO. Processing activity is propagated through the network as "trigger" data items are processed, generated, and communicated between related PO's.

The implementation of this concurrent specification of an application task, as sequential program structures on a single tasking microcomputer, required an inversion design transformation of the specification. The inter-PO data message flow between concurrent objects was transformed, by an inversion of the data flows, into parameter passing at sub-module procedure calls. The inversion implementation technique used in data-driven software design methodologies such as JSD and JSP, is documented in [Jackson83], [Cameron86], and [Storer88]. This design transformation obscured the parallelism inherent in the application model.

In P-DB4GL, the inversion transformation is not necessary. The concurrent specification of an application task is implemented using a concurrent programming language, Occam. A direct translation, from concurrently executable DB4GL objects to concurrent Occam processes, is possible. The inter-object message passing is implemented by inter-process communication using Occam channels. The concurrent Occam programs can be loaded onto transputer networks for parallel execution (Figure 5.2).

5.3 Development Approach

The development approach taken has been to construct a prototype P-DB4GL system. In the prototype system, simple database applications have been designed, implemented, and test run on different transputer networks. The practical experience gained from constructing a prototype implementation has been used to assess the suitability of Occam as an implementation language. Results obtained from test runs of the simple applications on transputer networks provide measurements of the transputer networks' performance under realistic loads, and have identified key areas of further development. Based on the data obtained from the prototype P-DB4GL implementation, designs for a fully functional P-DB4GL are proposed (see Chapter 6).

The test applications used in the prototype P-DB4GL have not been generated automatically from data dictionary specifications by P-DB4GL generation tools, but have been directly hand coded in the target programming language, Occam. The P-DB4GL generation tools will form part of the fully functional P-DB4GL system when improved code modules have been developed, and the exact mechanism of code generation established. The version of Occam used is the latest release, Occam 2 [Inmos88b]. The P-DB4GL software had initially been developed using the Transputer Development System (TDS) [Inmos88c] hosted on a PC, but the software has now been converted to run under the Inmos Toolset [Inmos89d]. All of the original DB4GL system had been written in COBOL, and though a COBOL compiler is not currently available for the transputer, it may be possible to re-use some of the original DB4GL code when a COBOL compiler becomes available.

5.4 Implementation

The prototype test applications do not implement the entire functionality of the fully developed DB4GL application model as described in [Hird89]. Some features, such as the integrity rule processing and involuted (or recursive) data relationships, have been omitted from the prototype implementation. The test applications are functionally equivalent to the applications generated by an earlier version of DB4GL described in [Ewin85a]. The original factoring and classification of DB4GL code modules into User Process modules and Data Access modules has been retained; but with separately compiled (SC) Occam processes replacing the separately compiled COBOL program modules used in the original DB4GL implementation.

The decision to retain the original factoring and classification of code modules in the prototype P-DB4GL implementation necessitated compromising the design methodology of a direct object to process translation. Consequently, the resultant translation from application model specification to concurrent process implementation is not so clear. The reasons for this decision and the implications are discussed in the next two paragraphs.

At the start of the P-DB4GL research project the DB4GL application model had not been fully developed to the level of sophistication described in

[Hird89]. The fully developed application model contains many sophisticated, and complicated to implement, features such as the integrity rule processing. In order to reduce the software engineering effort needed to implement a prototype P-DB4GL system, some of the functionality of the application model was discarded, and a version of DB4GL described in [Ewin85a] and [Poole87] was taken as the basis for the prototype P-DB4GL construction. The design translation performed in this earlier DB4GL version had to be first reconstructed by working backwards from the executable code modules and performing the design in reverse to arrive at the original specifications.

In a further attempt to reduce the software engineering effort, it was decided that parts of the prototype P-DB4GL implementation would be represented by simulations and test harnesses. Only the data access modules would be fully implemented. The resultant data access processes in the P-DB4GL implementation, that is, the prime and coupling entity handlers and the Filer and Disc processes, would have been present with the same functionality even if a more direct translation from Presentation Objects to concurrent process had been performed. Many of the modules in the original factoring, for example, the User Process (Screen and Window) modules, which have no correspondence in a direct translation were not present in the prototype P-DB4GL system, they were only simulated. Consequently, the most important results obtained from testing the prototype, concerning data access process performance, would have been substantially the same even with a direct design translation (and partial simulation). In particular, the conclusions drawn about processor loading, communications/processing ratio, and message passing overheads, can with justification be applied to an implementation based on a direct Presentation Object to concurrent process design translation.

A Presentation Object (PO) is not implemented by a single (SC) Occam process. Each Information Unit (IU) is implemented by an Entity Handler process. An application task's data access schema (the IUG and schema link processing) is implemented by a Schema Handler process. The Presentation Units, Process Schemas, and Process Tasks of the constituent PO's are realised by a number of User Processes. The propagation of processing activity through the network of PO's in an application task is realised as message passing on the Occam channels that connect the separately

compiled Occam processes which constitute the implemented application task.

The P-DB4GL applications have not been designed for execution on a specific transputer network topology, although some general features are assumed. The transputer network contains a large number of small (typically 20-50 Mbyte capacity), transputer controlled, Winchester hard discs. Each disc stores only one, or at most very few, of the files required by any particular application. Within the network, some transputers (normally the majority) will be designated as Filing nodes, some transputers will be Processing nodes, and some (usually one) will be User Interface nodes (Figure 5.3). One of the objectives of the prototype P-DB4GL implementation has been to experiment with different network topologies and mappings of concurrent program to these topologies.

A uniform communication protocol for P-DB4GL inter-object message passing has been used that accommodates both control and data messages. This protocol, known as the Basic Communication Unit (BCU), is modelled on the BCU described in [Ewin85a] and [Hird89] for DB4GL inter-object message passing. In DB4GL, message passing is achieved by parameter passing between calling and called code modules. In P-DB4GL, this is replaced by a two-way Request-Reply protocol (Figure 5.4): if process1 wishes to send a message (control or data) to process2, a BCU-Request is sent to process2 suspending process1's execution; after reception and processing of the BCU-Request, process2 sends a BCU-Reply to process1; upon receipt of BCU-Reply, process1 continues its execution. The BCU message packet is composed of a number of fields identifying: the type of information contained (data or control); the source and destination objects; the error status of the message request; the data length and data (if any). The full message format and the Occam protocols for BCU channels is given in Appendix B.

As with many database applications it is the disc access that is the limiting factor to overall system performance, in DB4GL the processing time of the user modules is negligible compared to the time it takes the data access modules to retrieve and store records. It is for these reasons that the data access processes have received the most attention in the attempt to gain maximum benefit from the parallel processing hardware. The data access processes (Schema handlers and Entity handlers) have been designed to

provide the same functionality as the original (sequential) DB4GL data access modules, but they incorporate concurrent algorithms which can provide improved performance using the parallel transputer hardware. Figure 5.5 shows where the main benefits of concurrent data access lie in the P-DB4GL applications:

- concurrent schema processing, the schema handler process can be processing data access messages whilst its entity handlers independently process and access their files;
- concurrent entity processing, all the entity handler processes can access their files in parallel when the files are stored on separate discs, thus increasing disc throughput;
- concurrent coupling update, a prime entity handler can update its coupling entity handlers in parallel;

A number of test harnesses and simulations have been used in the implementation of the P-DB4GL test applications (Figure 5.6). The User processes, that "drive" the data access processes have not been fully implemented. In the P-DB4GL test applications, a User Process Test Harness (User Harness) has simulated the behaviour of the User Processes. The User Harness delivers a stream of BCU request-reply messages to the data access processes. This message stream, which incorporates variable length delays between messages, is representative of the interaction between genuine user processes and data access processes in a P-DB4GL application. A full description of the User Harness is given in Appendix F. A simulation of a multi-disc parallel filestore has been used to support the filing requirements of the test applications. The simulation is composed of Disc and Filer processes. It is possible to alter both, the number of simulated discs, and the distribution of files to discs (Figure 5.7). The Filer process provides the functions of a single Index-Sequential file of records. The Disc process incorporates delays simulating mechanisms such as head movements. The Disc and Filer processes are described in detail in Appendix D. In a fully functional P-DB4GL system, genuine filing processes and transputer controlled discs will replace this simulation.

P-DB4GL has a coarse grained level of concurrency. In P-DB4GL, the unit of distribution, suitable for allocation to a processor, is the separately compiled (SC) Occam process equivalent to a separately compiled code module of DB4GL (ie Entity handler, Schema handler, User process). A processor may be allocated more than one SC process, but usually an SC process cannot be

decomposed into further levels of concurrent execution. Where a P-DB4GL SC process does contain nested concurrent process, these processes cannot be distributed across multiple processors.

Occam programs, when executed on transputer networks, only permit a static allocation of code to processor. Before a P-DB4GL test application can be executed on a network, it has to be configured. The constituent SC processes have to be allocated to processors, and logical communication channels between processes have to be mapped onto physical point-to-point serial links. This configuration information can be altered without recompiling the SC processes of the program, and different configuration can easily be tested on a network. The P-DB4GL test applications have been executed on a variety of networks with many different configurations. Because the test applications contain a small number of SC processes, the mapping of channels to links has been simple and straightforward. As the number of processes in a program increases, the mapping of channels to links becomes progressively more difficult, eventually there are more channels than links available and it becomes necessary to multiplex channels over links. This has not been necessary in the configuration of the P-DB4GL test applications, but it is an issue that is addressed in the designs for a fully functional P-DB4GL system.

5.5 Testing

Each P-DB4GL test application consists of:

- a User Harness and filing simulation;
- a number of data access processes (Schema and Entity handlers) that constitute a data access test schema;
- test data;
- network configuration information.

For each test application several test runs have been conducted, in which the User Harness behaviour and filing simulation parameters have been varied. The test runs have been conducted on different network configuration with between one and five transputers. A typical configuration for a test run is shown in Figure 5.8. A detailed description of the test schemas and configurations can be found in Appendix G.

The User Harness supplies test data to the test application, and can be operated in either interactive or batch modes. In interactive mode, single

BCU Request-Reply messages are sent to and received from the data access processes. When used in batch mode, a sequence of transactions is performed on the data access processes. Each transaction is composed of a number of BCU messages, which together perform some complete database action such as storing or updating a record. The term "transaction" is not being used in the normal database sense of an indivisible collection of updates. Currently, P-DB4GL is a single user database system, concurrent processes within a P-DB4GL application do not interfere with each other and should not be able to deadlock, there is no facility to recover should a transaction fail. The decomposition of database actions (transactions) into a number of smaller "atomic" actions (each one represented by a BCU message) allows optimizations to be performed in a sequence of transactions, thus reducing the amount of data communicated between the P-DB4GL objects. Appendix F provides an example of optimizing the BCU messages in a sequence of transactions. Execution times for individual BCU messages, transactions, and complete test runs, are recorded by the User Harness. These timings are used to compare the performance of different test applications.

The reason for performing the many different test runs is to determine the effect of two things:

- concurrent vs sequential algorithms - how does the performance of concurrent data access algorithms compare with sequential algorithms, for a test application running entirely on a single transputer;
- multi-processor configurations - what is the difference in performance when the data access processes (concurrent algorithms) are run on more than one transputer.

A number of functionally equivalent versions of the data access processes have been implemented. These versions differ in the degree of concurrency of their algorithms. Four versions of entity and schema handlers are briefly described below, full descriptions are given in Appendix C.

- 1 "original" sequential versions (v1) - the data access processes repeatedly: receive a BCU Request message; process that request; and return a Reply message. Processes are idle in the intervals between BCU messages.
- 2 "modified" sequential versions (v2) - as v1 handlers, but processes can return a BCU Reply message and then be kept busy with

- processing activity during the interval before the arrival of the next BCU Request message.
- 3 concurrent handler versions (v3) - as (v2), but the processing performing in the intervals between BCU messages includes concurrent algorithms (ie the use of Occam PAR constructs).
 - 4 concurrent handler versions (v4) - as (v3), but with improved concurrent algorithms, for example, parallel coupling entity update by prime entity handlers.

5.6 Results

Results obtained from the many test runs performed on the P-DB4GL test applications are presented in detail in Appendix H. The version 2 data access handlers typically produce a 5%-20% reduction in execution time over the version 1 handlers for a variety of test data. The maximum improvement factor (or speed-up) available is 1.30. The exact improvement for any given application depends both on the test data used, and the run time behaviour of the application, that is, the frequency and duration of inter-BCU and inter-transaction processing delays. The version 3 handlers are typically 15% to 30% faster than version 1 handlers, with a maximum improvement factor of 1.53.

Results for version 4 handlers show the most significant improvements (see Figures 5.9, 5.10, 5.11). On test schema SCHM0004 with three coupling entities, version 4 entity handlers are typically 5%-40% faster than version 3, and show a maximum improvement factor of 1.72. As the number of coupling entities updated is increased, the improvement factor for version 4 handlers over version 3 handlers increases proportionately; so that, for SCHM0004 with 18 coupling entities an improvement factor in test run time of 6.29 is attained. When the time taken for coupling entity update is taken alone, the improvements are even greater. The improvement factor for coupling update only ranges from 2.38 (for three coupling entities) to 9.14 (for 18 coupling entities). For a version 3 entity handler, the time taken to update N couples is nearly N times greater than to update one couple; but, for a v4 entity handler, the time taken to update N couples is almost the same as the time to update one couple.

These dramatic improvements in execution time for the concurrent coupling entity update (v4 entity handlers) have been obtained with all the

data access processes executing entirely on a single processor. Tests have also been conducted with identical programs running on multi-transputer configurations, but the execution times of the test runs are practically the same as the single processor configuration. The additional available processing power of the multi-processor configurations has not effected the performance of either the concurrent or sequential version of the data access processes.

The reasons for these effects lie in the proportions of processing, idle, and communication times present in the test runs. The disc access time is orders of magnitude greater than processing and communication time. During disc access, a processor running the sequential handler algorithm is idle, thus, the total test run time is largely composed of a sequence of disc access delays. For a processor running the concurrent handler algorithm, the disc access delays occur concurrently, thereby reducing the total test run time (see Figure 5.13). However, as the number of coupling entities updated is increased, the ratio of processing time to idle time increases, and the improvement factor of the concurrent algorithm over the sequential algorithm declines (see Figure 5.12). In the test runs conducted so far, tens of coupling entities have been concurrently updated, resulting in small processing loads for a single processor. In large applications, with hundreds of concurrently executing data access processes, a single processor would become significantly loaded, and benefits of multi-processor configurations will become evident (see Figure 5.14).

5.7 Evaluation

The prototype Parallel-DB4GL (P-DB4GL) system has shown that a transputer-based implementation can provide significant performance improvements for the unchanged DB4GL data model. The dramatic improvements in test application performance, the coupling entity update (version 4 handlers) in particular, demonstrate the advantages of a target programming language (Occam) that supports the implementation of concurrent data access algorithms. However, the prototype P-DB4GL system falls far short of a fully functional P-DB4GL system; the applications have used simulations and test harnesses which would not form part of such a fully functional system. The P-DB4GL test applications have mostly been test run with small applications and all of the application code (filing simulation included) executing on a lightly loaded single processor. If the

prototype P-DB4GL system is to be developed into a fully functional multiprocessor P-DB4GL system, there are some important issues to be addressed and several key areas requiring further development work, these are described below.

It is not possible to directly compare the absolute performance of the sequential COBOL implementation of DB4GL on a PC, with the performance figures obtained from the prototype P-DB4GL test applications. Unlike the full sequential implementation of DB4GL on the PC, the P-DB4GL test applications only partially implement the functionality of a DB4GL database application; the data access processes are properly implemented, but user processes have only been simulated. The P-DB4GL test applications use a simulation of a parallel filestore, which does not directly model the behaviour of any genuine filing system. The performance of the sequential DB4GL applications are to a large extent determined by the performance of the PC's disc drives and the run-time COBOL environment's filing software, and these cannot be directly compared to the test applications' filing simulation.

The purpose of the P-DB4GL filestore simulation is to allow comparisons of the relative performance of the different versions of the data access processes (with differing degrees of concurrency in their algorithms). In a fully functional multiprocessor P-DB4GL system the filestore simulation will be replaced by genuine discs and filing processors, and for this reason, inferences about the absolute performance of such a fully functional P-DB4GL cannot be made on the basis of figures obtained from the test applications. The sequential DB4GL implementation had not been developed with performance in mind; consequently, there are no benchmark performance figures with which to compare the performance of a fully functional P-DB4GL. When a fully functional P-DB4GL is implemented, it should be benchmark tested along with the original DB4GL for the purpose of speed-up, scale-up, and price/performance comparison.

The P-DB4GL test applications (as described in Appendix G) are very small and unrepresentative of genuine (P-)DB4GL database applications. The largest test schema used was SCHM0004 with one prime and 18 coupling entity handlers - this limit was fixed by the 2 Mbyte memory size of the B004 transputer boards used for the test runs. The size and number of the attributes in the Test Entities (Appendix Figure G1) was untypically small,

generally about 10 bytes per record with only a few tens of records per file. The user processes were only simulated, and the User Harness simulation did not generate a processing load comparable to genuine user processes. A genuine P-DB4GL application in a fully functional system would be significantly larger in terms of: data sizes; numbers of constituent processes; and numbers of files and discs. The effect of these changes would be, to increase the total processing load produced by an application, and make the connection of discs to data access code somewhat difficult. It is not altogether obvious how the many (possible hundreds) of discs, assumed in the filestore simulation, could easily be connected to an application executing entirely on a single processor.

Distributing a large application across several processors would make the task of disc connection easier. Also, when a large processing load is shared amongst a number of processors, the benefits of multiprocessor speed-up are likely to appear (Figure 5.14). But, distributing an application across a multiprocessor network makes software configuration, particularly channel connection and placing, more difficult; and in Chapter 6 (Designs for a Fully Functional P-DB4GL) the attendant problems of channel multiplexing and message routing are examined. Another aspect to distributing an application is the increased opportunity for performance optimization using various well documented [Atkin89] hardware and software modifications. Some attempt at this performance optimization was made with the P-DB4GL test applications and is described in Appendix I; but with such small programs and single processor configurations, the opportunities were limited and the results were somewhat disappointing.

An important feature missing from the prototype P-DB4GL system is the DB4GL data dictionary. One of the tasks involved in designing a fully functional P-DB4GL system will be the construction of an enhanced P-DB4GL data dictionary. The data dictionary schema (DDS) will be similar to the DDS of the original DB4GL [Ewin85a], but will contain additional information such as:

- the hardware configuration available to P-DB4GL;
- the mapping of entities to discs;
- the typical processing requirements and distribution of code to hardware for each application.

This information is needed for the generation of P-DB4GL database applications. When P-DB4GL is used as a database application prototype

tool, this information will be updated by the system designer as new applications are defined and existing applications are improved and regenerated.

In conclusion, DB4GL database applications, in common with many other database applications, tend to be disc input-output (I/O) bound. The main benefit of the transputer-based implementation lies in the ability to perform multiple concurrent disc I/O, thus increasing disc I/O throughput, and hence reducing overall application execution time. An important feature of DB4GL is the reliance on indexes (coupling entities) to support pre-compiled queries with fixed access paths through the database. The maintenance of these coupling entities following an update to the prime entity to which they are coupled, produced processing delays significantly impairing the performance of the original sequential DB4GL. In P-DB4GL, the coupling entities are processed in parallel, the delays associated with maintaining these coupling entities are eliminated, thus improving application execution time. Other database systems dependent on heavily indexed relations, might also benefit from a similar parallel implementation, in which index maintenance can be performed concurrently.

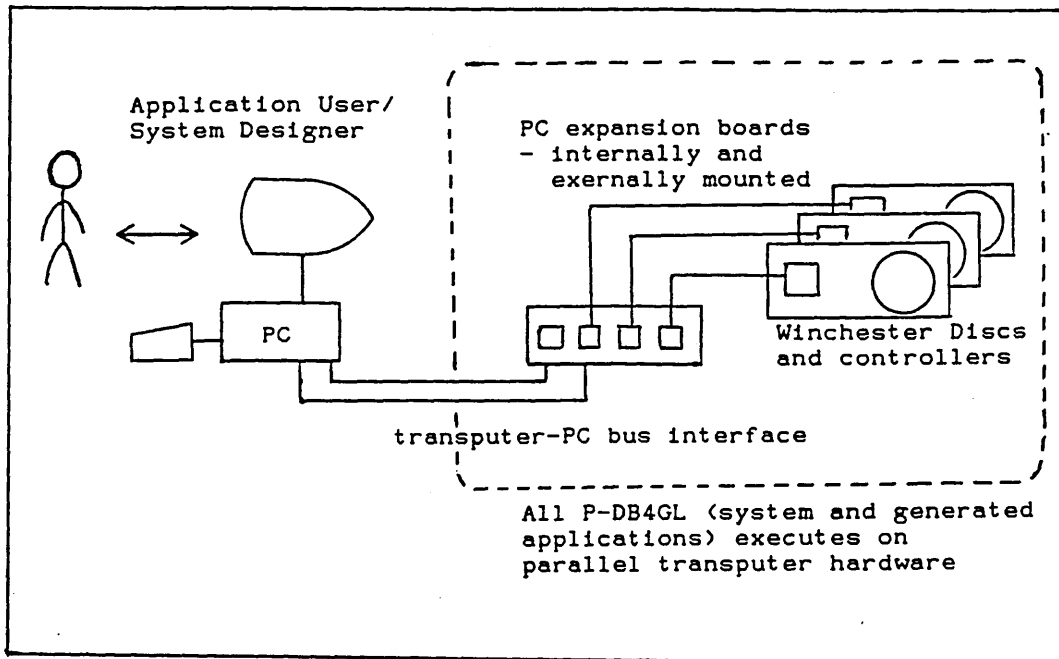


Figure 5.1 - System Configuration for P-DB4GL

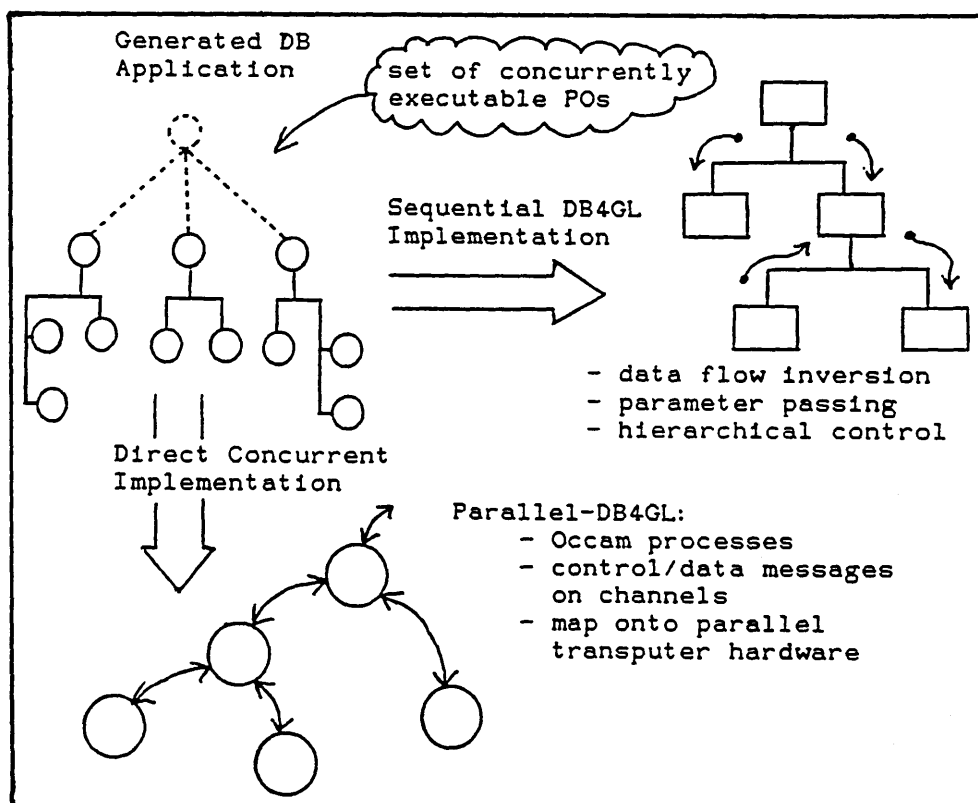


Figure 5.2 - Concurrent Message Passing Processes

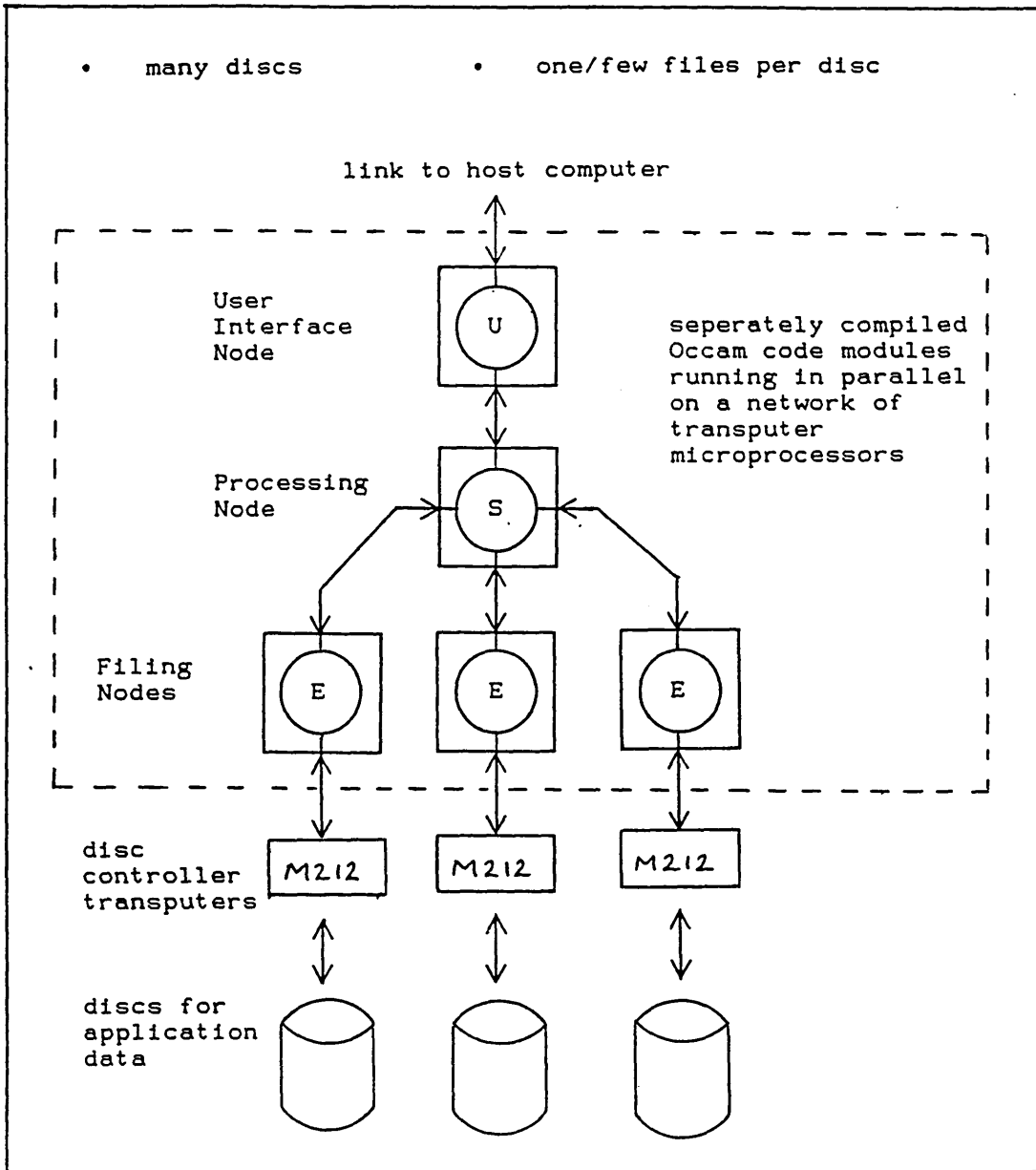


Figure 5.3 - Typical P-DB4GL Database Application

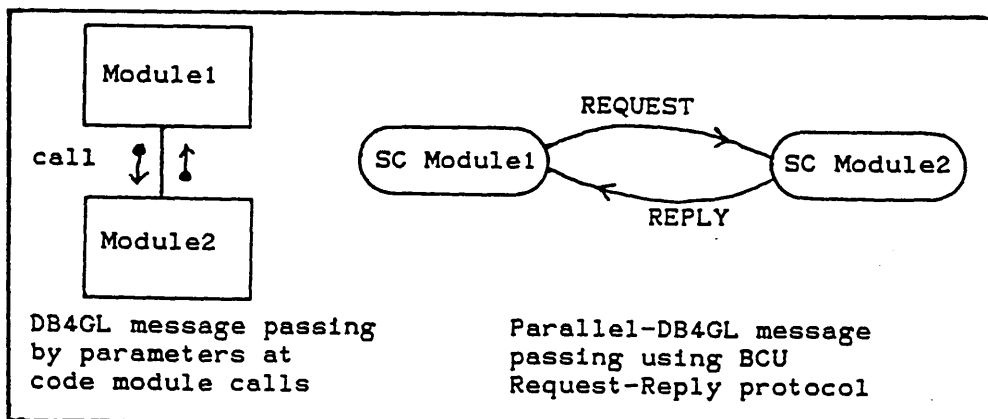


Figure 5.4 - DB4GL and P-DB4GL Message Passing

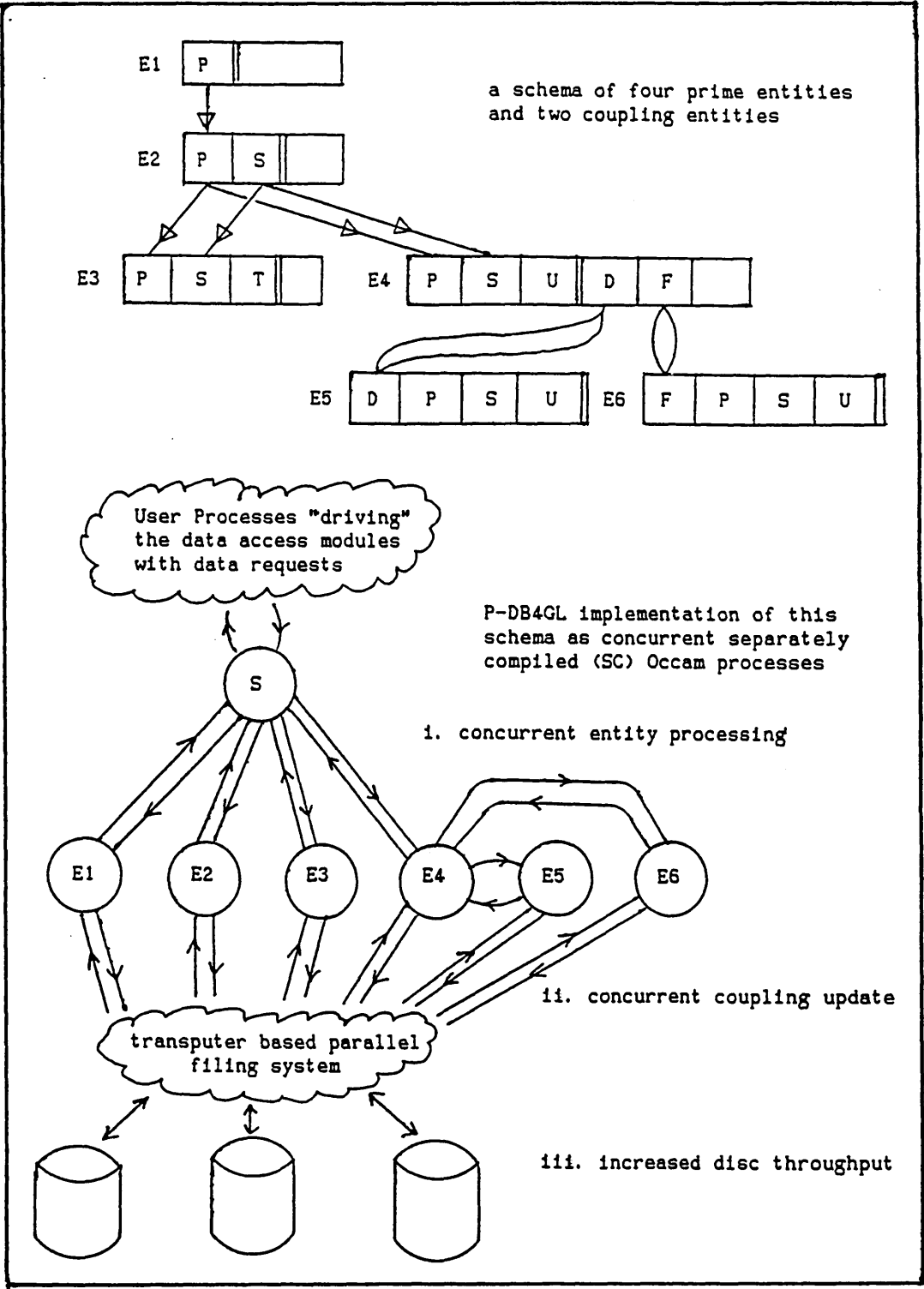


Figure 5.5 - Concurrent Execution of Data Access Processes

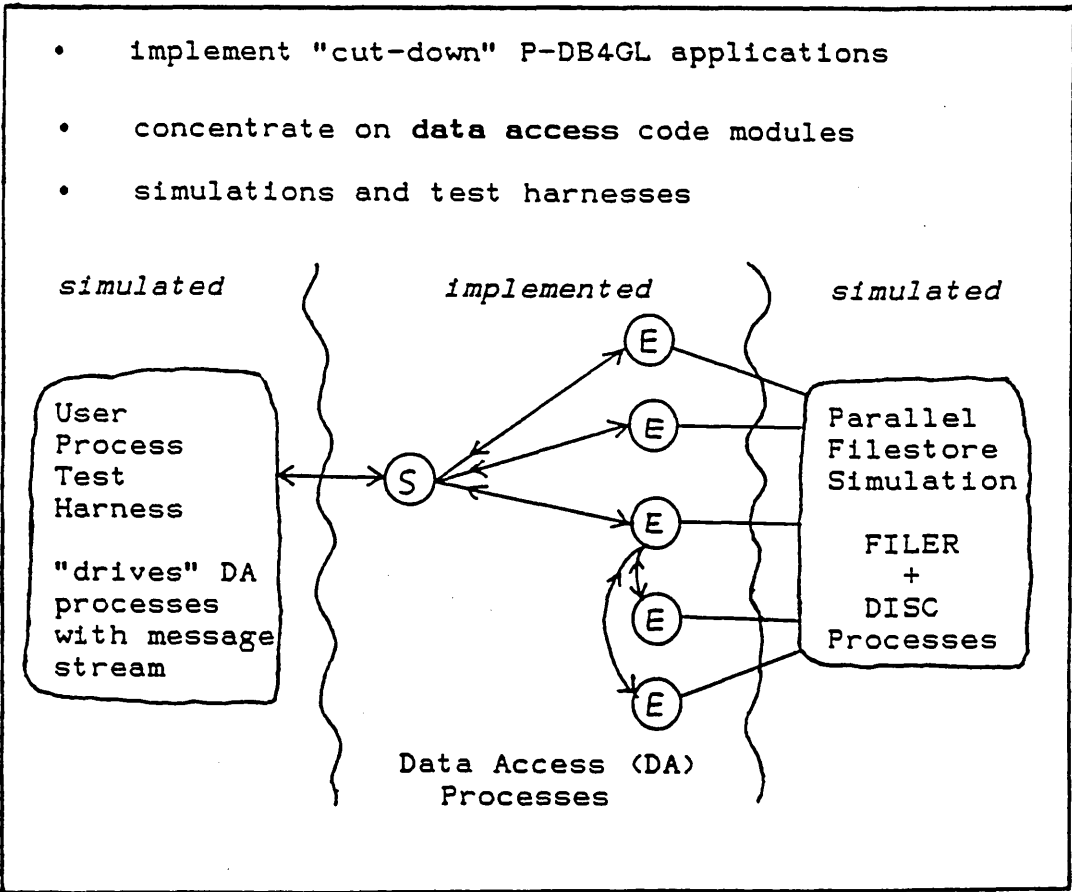


Figure 5.6 - Developing the Prototype P-DB4GL System

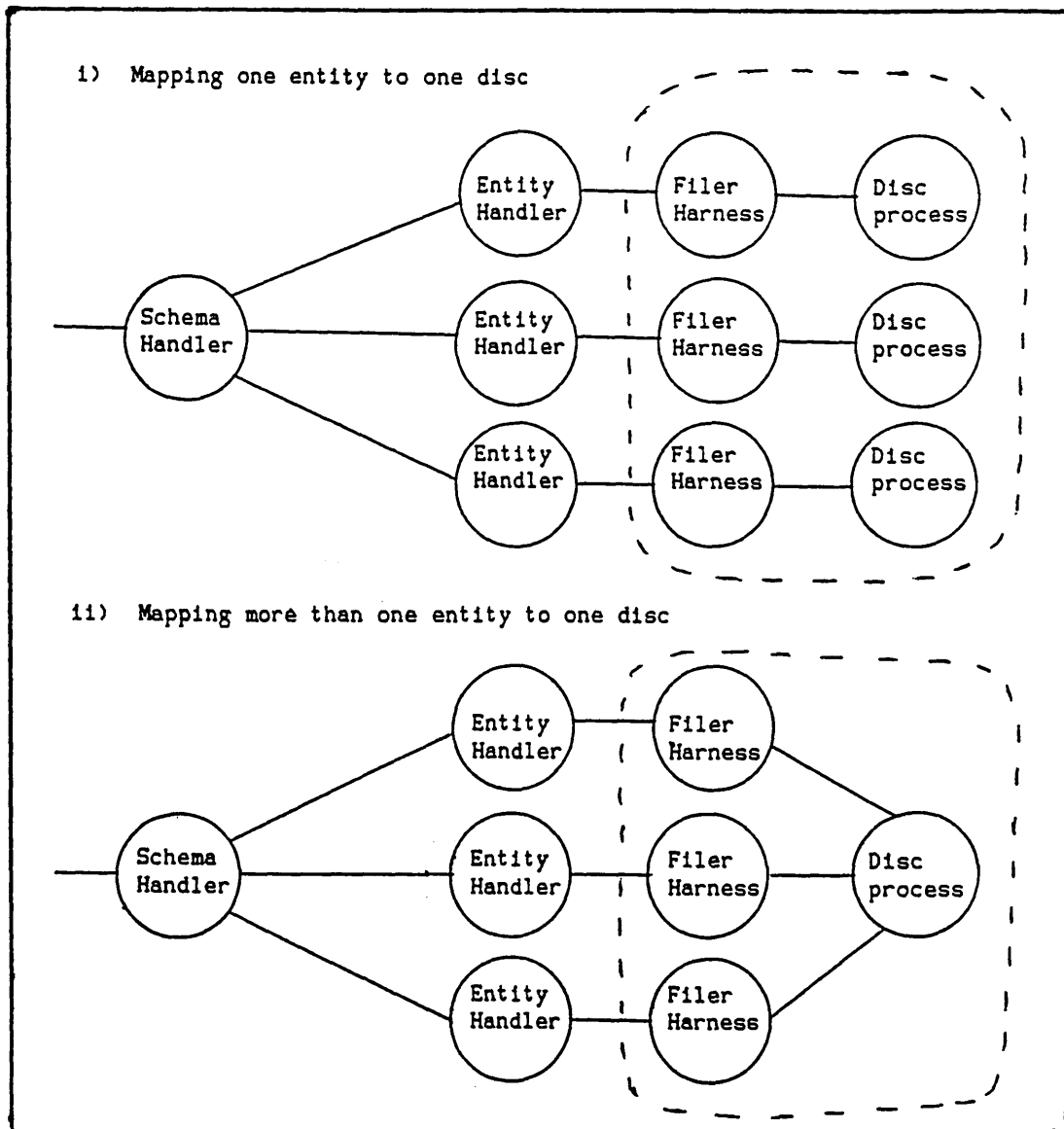


Figure 5.7 - Mapping Entity Handlers to Discs

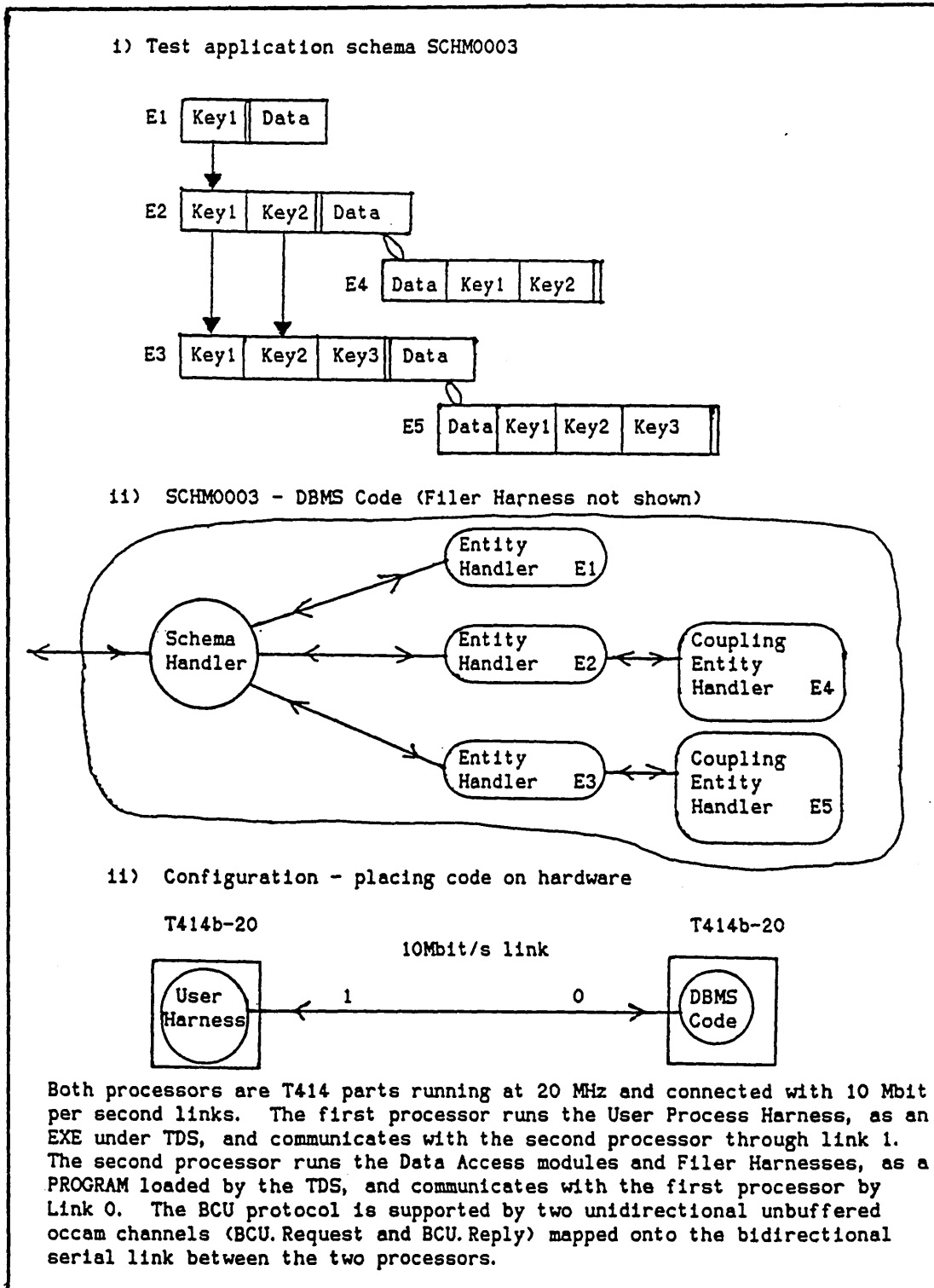


Figure 5.8 - Typical Test Application Configuration

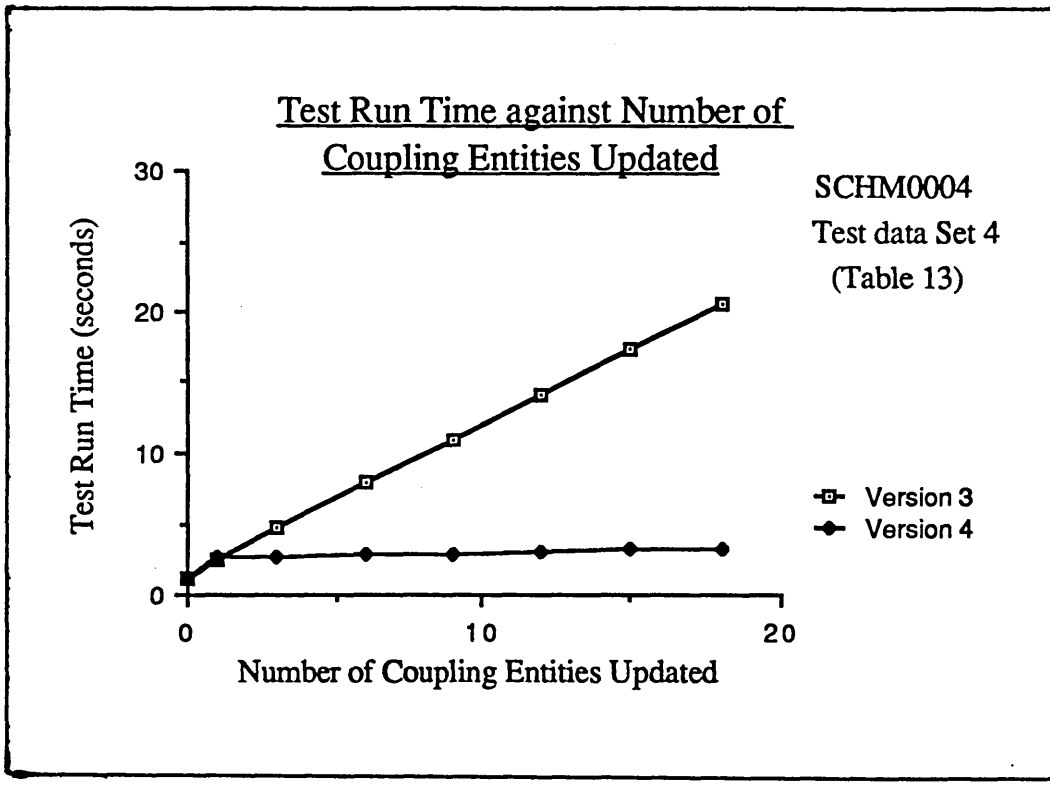


Figure 5.9 - Version 4 Handlers Test Run Times

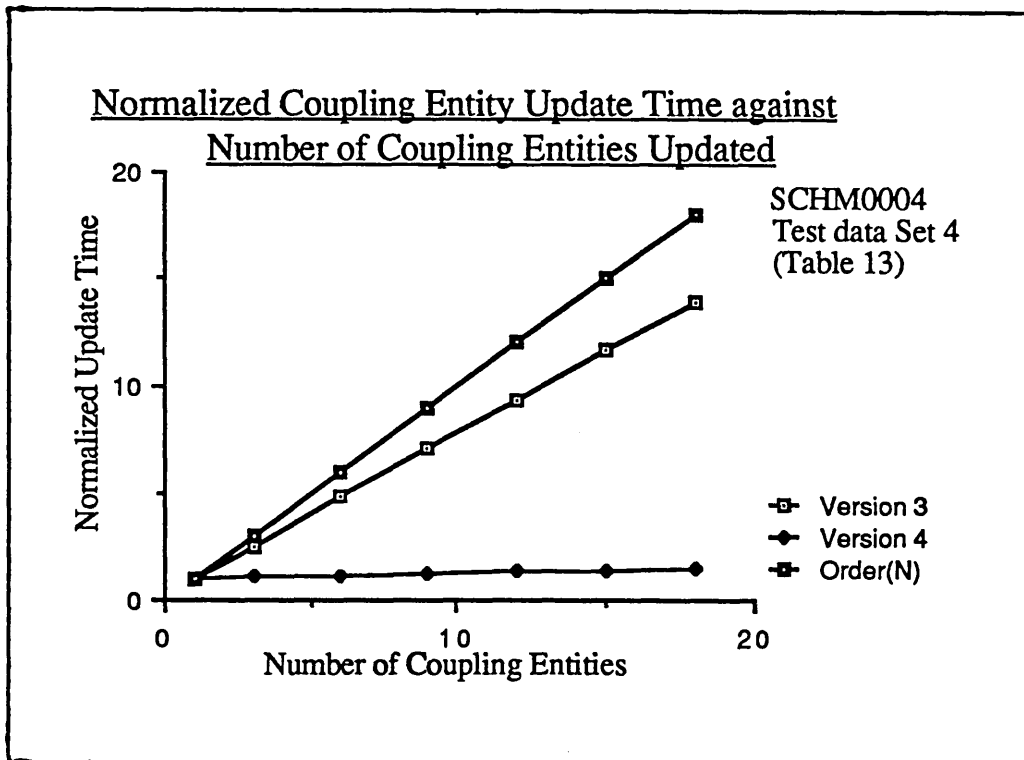


Figure 5.10 - Normalized Coupling Entity Update

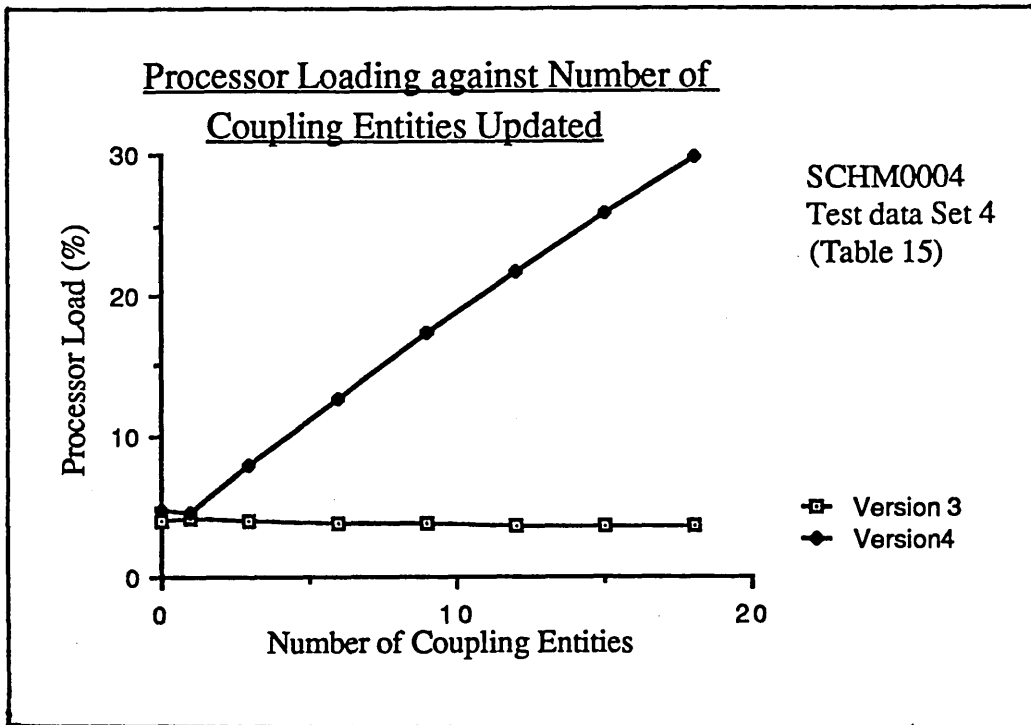


Figure 5.11 - Version 4 Handlers Improvement Factor

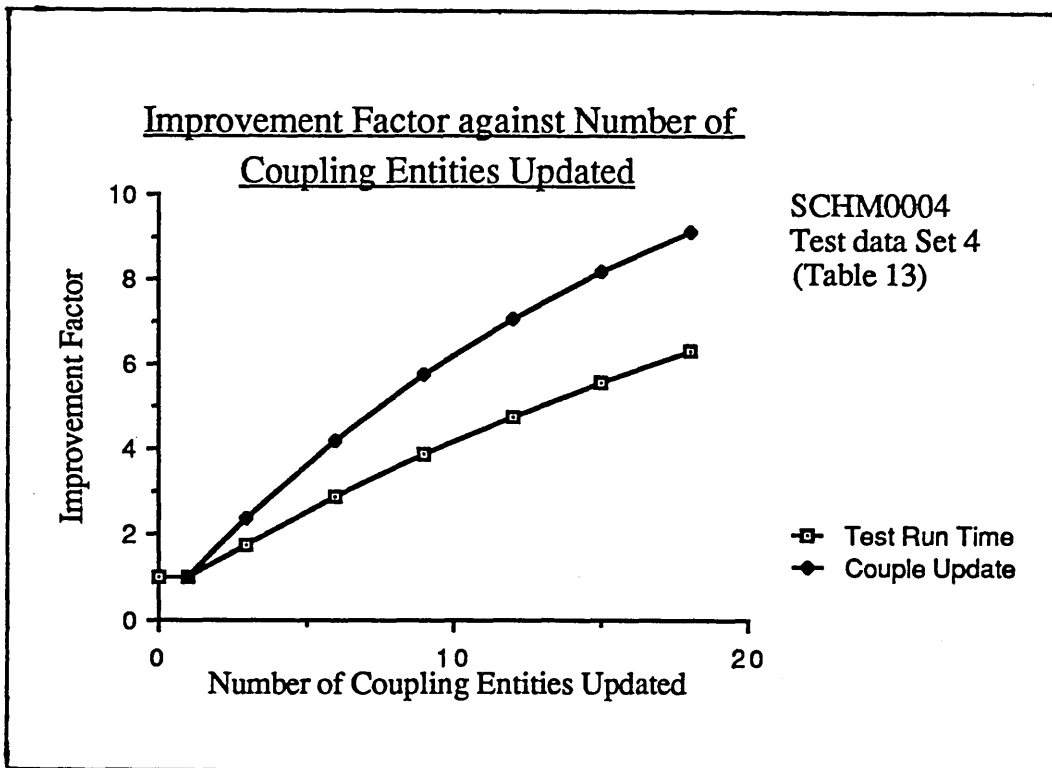


Figure 5.12 - Test Run Processor Loading

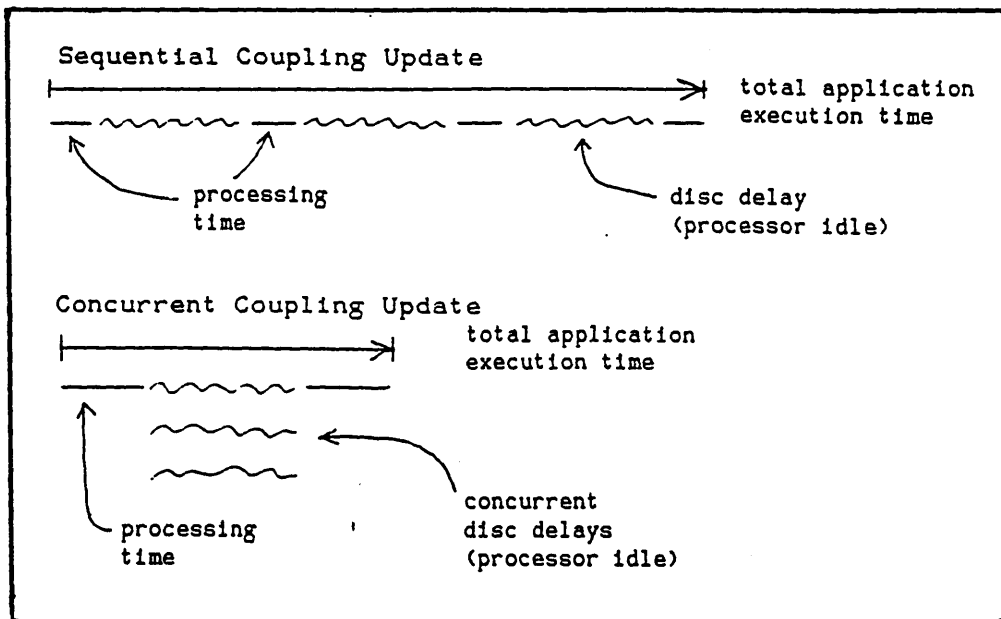


Figure 5.13 - Processing/Disc Access Ratio

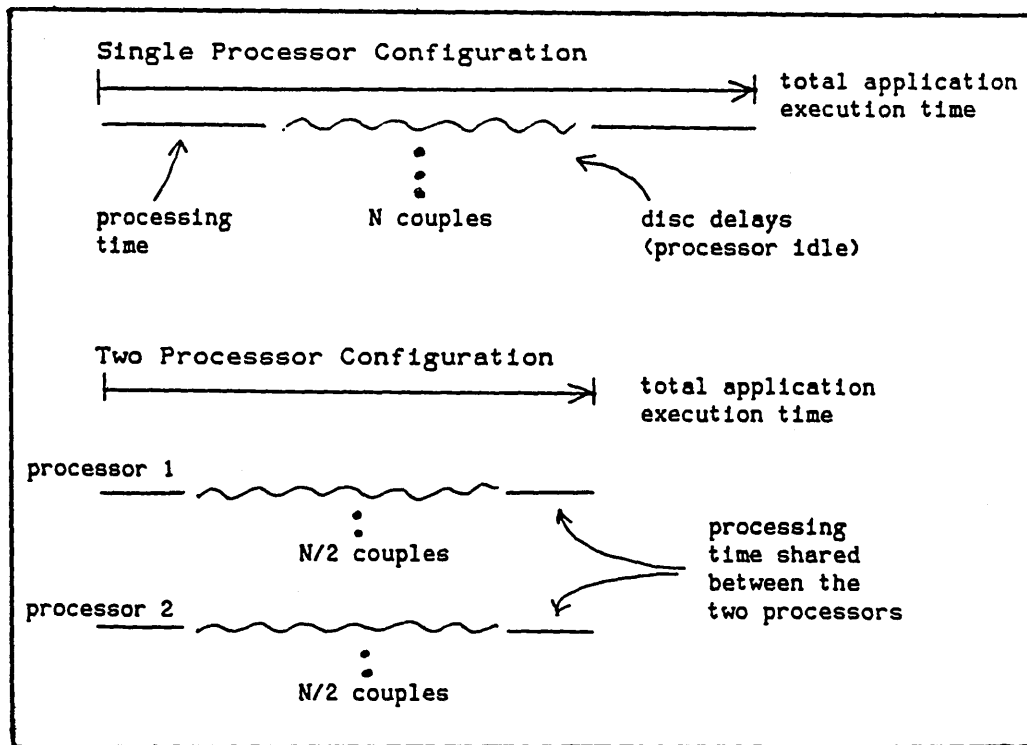


Figure 5.14 - Processor Loading and Parallel Configuration

Chapter 6

Designs for a Fully Functional P-DB4GL

6.1 A Multiprocessor Fully Functional P-DB4GL

The significant performance improvements identified in the prototype P-DB4GL implementation are based on a simulation of a parallel filestore. In the small P-DB4GL test applications, the data access code and filing simulation execute on the same processor; therefore, the task of connecting entity handlers to their Filer processes in the filing simulation is quite simple. However, if in a fully functional P-DB4GL the filing simulation is to be replaced by genuine filing software and multiple discs and controllers, then connecting entity handlers to Filer processes becomes a problem. In a realistically sized P-DB4GL database application (with perhaps ten to a hundred concurrent entity handlers), if all the data access processes execute on a single processor, then it becomes very difficult to connect the many separate filing processors to the data access code processor (Figure 6.1). The transputer microprocessors have only four serial links, and each link can support two Occam channels. With only one data access code processor and many filing processors, there simply would not be sufficient links to support the channels connecting the entity handlers to their filing processes.

A solution to the entity handler/Filer connection problem is to distribute the data access code across several processors. Each data access code processor can then be directly connected to four filing processors. However, the distributed data access code processes are connected both to each other, and to P-DB4GL code such as the User processes, by more Occam channels that also require transputer links to support them. Consequently, not all of the data access code processors' links are available for connection to filing processors, some links must be used to interconnect the data access processors themselves (Figure 6.2). Because the exact pattern of channel connections between code processes varies from one application to another, a hardware configuration (the link connections in particular) suitable for one application is unlikely to be suitable for others. It is therefore necessary to either: reconfigure the hardware in between application runs; or use a fixed hardware configuration, but alter the application code to execute on the chosen fixed configuration. In the initial designs for a fully functional P-DB4GL the second approach was taken. The hardware is configured in a ring topology, and the application code is modified by the addition of a Kernel

process placed at each processor in the transputer network (Figure 6.3). The Kernel process is responsible for invoking and initializing the component processes of the distributed application allocated to its node in the network. The Kernel process is also responsible for routing messages between the connected processes of the distributed application.

In the remaining sections of this chapter, this initial design is examined in more detail and some alternative designs are proposed. The message routing function of the Kernel has received the most attention as far as development work goes: a P-DB4GL Router process has been implemented and tested in order to assess the communication overheads involved in message routing. The mechanism of code generation, and the factoring of the application code, are also examined and alternative solutions suggested.

6.2 Channel Multiplexors and Message Routers

6.2.1 Channel Multiplexing

A problem which often arises when concurrent Occam programs are executed on transputer networks is the mismatch between the number of channels in a program and the number of physical links in the network. Before an Occam program can be executed on a transputer network, software configuration information is added to the program - placing processes on processors, and mapping channels between processes on separate processors onto connecting physical inter-processor links. A major constraint on the software configuration is the restriction of a maximum of only two unidirectional channels (one in each direction) placed on each bidirectional physical link. For Occam programs with a large number of highly interconnected processes it can sometimes be difficult, or even impossible, to find a software configuration that matches the available hardware (Figure 6.4). Even when such a match exists, it is often found that the matching software configuration results in a poorly balanced processing load on the network with consequent lack of multiprocessor performance speed-up.

A solution to this channel placement problem is the addition of channel multiplexor processes to the Occam program (Figure 6.5). When the software configuration is altered in order to experiment with alternative load balancing, the multiplexor processes can be added or deleted as appropriate. However, the implementation of such multiplexor processes

and the incorporation of them into Occam programs is an additional burden on the programmer. The multiplexors may in certain circumstances alter the behaviour of the program by inadvertently buffering communication and thereby interfere with process synchronization. The execution of multiplexor processes is an additional processing load on the processor, the time taken to multiplex and decode channel messages can cause delays affecting program performance. Also, the amount of data communicated on several multiplexed channels could exceed the data transfer rate of a single physical link.

6.2.2 P-DB4GL Channel Multiplexors

In the prototype P-DB4GL test applications the number of Occam processes is relatively small (generally tens, rather than hundreds of processes), and the size of transputer networks upon which they have been executed are small too (typically two to four processors). Nevertheless, even with such small programs and networks, the constraint of two unidirectional channels per physical link has restricted the variety of test configurations used. In a fully functional P-DB4GL with hundreds or even thousands of processes and significantly larger transputer networks it is envisaged that software configuration, in particular the mapping of channels to links, will be an even more difficult task. To aid the development and testing of fully functional P-DB4GL application code, and facilitate simple software reconfiguration, channel multiplexor processes have been implemented for the P-DB4GL data access processes.

The P-DB4GL channel multiplexing involves two stages. At the first stage, the many BCU channel pairs connecting each data access process to other application code processes are multiplexed into a single pair of channels conforming to the BCU-extended (BCUX) protocol defined in Appendix B. The BCUX protocol extends the BCU message format with additional fields indicating the identity and object type of the source and destination processes of each BCU message. Figure 6.6 illustrates this stage of channel multiplexing for a modified prime entity handler, and shows how the schema handler request-reply channel pair and the coupling entity handler request-reply channel arrays are multiplexed together and converted to the extended protocol. The BCUX multiplexor process inside each modified data access process uses the information contained in these additional fields to

decode incoming BCUX messages and direct them to the appropriate ingoing BCU channel.

At the second stage of channel multiplexing, all the BCUX channel pairs belonging to the data access processes intended for execution on the same processor are connected to a single Decoder process (Figure 6.7). The Decoder process possesses a look-up table containing the object identity of each data access process connected to it. In addition to the pair of BCUX channel arrays connecting the Decoder to its data access processes, there is another pair of BCUX channels. This additional pair of channels is used to receive and send external BCUX messages to Decoder processes on different processors. When the Decoder receives a message from one of its own processes, it will attempt to redirect the message to the intended destination process; if the destination cannot be found in the look-up table, the message is forwarded on the outgoing external BCUX channel. Similarly, with messages received on the incoming external BCUX channel, the Decoder uses the look up table to redirect the message to one of its own processes. Incoming external messages which cannot be decoded are ignored. The Decoder and Multiplexor processes have no facility to correct or report errors found in BCU messages, it is assumed that messages sent by data access processes will always be properly received; if a message becomes corrupted or is incorrectly decoded, there is as no recovery action to such a failure.

The P-DB4GL Decoder and Multiplexor processes introduce buffering into the communication of BCU messages. For some Occam programs, buffering is an undesirable side effect, because synchronization between processes at points of communication can be lost. However, it is not a problem for P-DB4GL applications; because the higher level request-reply protocol of BCU message pairs, implemented on top of the Occam communication primitives, is used to synchronize P-DB4GL processes, and this is unaffected by buffering of either the request or reply component of the message pair.

6.2.3 P-DB4GL Message Routing

The P-DB4GL channel Decoder process has been implemented and tested with small applications. Although it provides adequate channel multiplexing and facilitates simple software reconfiguration for a small network of two processors, it is not suitable for larger networks in which messages have to traverse intermediate nodes. For larger networks an

additional Router process is required (Figure 6.8). Considerable amounts of research on message routing for transputer networks already exists, and many different algorithms for a variety of network topologies can be found in the literature, for example [Roscoe87] [Qiang88] [Briat89] [Knowles89] [Roebbers89] [Gallizi90]. The prototype P-DB4GL test applications were not constructed for any particular network topology, and in designing a fully functional P-DB4GL several different network topologies, both regular and irregular, have been considered. One of the regular network topologies considered for a fully functional P-DB4GL is a ring topology, and a Router process for a bidirectional ring has been implemented and tested.

The P-DB4GL ring Router process uses a "slotted ring" algorithm with variable size slots and each node on the ring "owning" a slot, or data packet, on the ring. It is based on the deadlock-free ring routing algorithm given in [Welch89]; but the P-DB4GL Router process differs from Welch's processes in some respects, particularly in the buffering of messages entering and leaving the ring. The P-DB4GL Router process and the BCUX-ring (BCUXR) protocol defined for it, along with details of the testing procedures and performance figures for the Router process, are described in Appendix J.

Results obtained from testing the P-DB4GL Router and Decoder processes under simulated application loads, indicate that the communication overheads incurred by multiplexing, decoding and routing BCU messages could impair the performance of P-DB4GL applications when executed on ring topologies of small size (four to eight node rings). An average communication time for a BCU request-reply message pair sent between two data access processes on separate processors in an 8 node ring is approximately 2 to 5 milliseconds. Because of the attribute-oriented nature of BCU messages, a significant database action such as the update of a coupling entity handler by a prime entity handler may require the communication of tens of BCU message pairs. Thus, the communication overheads of such an operation are approaching the same size as the disc access times.

The significant improvements in execution times for concurrent data access processes obtained from the prototype P-DB4GL test applications, rely on the assumption that communication times (of BCU messages) and processing loads are small compared to disc access times. Alterations to the relative sizes of communication, processing, and disc access may negate the benefits

of the concurrent data access processes. It is necessary to identify the points of high communication levels within the concurrent P-DB4GL applications, so that high communication processes can be placed physically close together (preferably on the same processor). The ring topology is not suitable for larger networks because message distance increases proportionately to the number of nodes. Further investigation is required in the areas of communication balancing and choice of network topology for a fully functional network version of P-DB4GL.

6.3 Communication Loads and Protocol Overheads

The types of messages communicated between the different constituent processes of a P-DB4GL application varies greatly. There are significant differences in terms of size, frequency, and protocol overhead of the messages sent between different types of P-DB4GL process (Figure 6.9). These differences affect the communication loads on the channels interconnecting the application processes; and are therefore an important constraint when applications are configured for execution on multi-processor networks. The messages communicated between entity handlers and their Filer processes, on channels defined by the FILER protocols, are generally record-based (ie each message contains a whole record or a record key). The Filer processes are themselves connected by further channels to low level disc driver processes executing on disc controller transputers, where each message transfers a whole block of records. The BCU messages communicated between the P-DB4GL data access and user processes are generally attribute-based; and the transfer of a complete record from one process to another requires the communication of several separate BCU request-reply message pairs.

The number and frequency of messages communicated on the many different channels during an application run, varies considerably. For example, in test runs on SCHM0003 using test data Set 3 (see Appendix G), 177 BCU request-reply message pairs are transferred on the channel connecting the User Harness and the schema handler process. However, during this same test run, the schema handler communicates 297 BCU message pairs with the ENTY0001 entity handler. The other entity handlers in this test schema, ENTY0002 and ENTY0003, receive a similar number of BCU messages from the schema handler; so the channels connecting the schema handler to the entity handlers carry large numbers of messages

during an application run. Of all the BCU messages received by an entity handler during a test run, only a fraction of them will invoke message communications on the FILER channels. For example, with test schema SCHM0004 and test data Set 4 (Appendix G), the prime entity handler ENTY0004 receives 87 BCU messages of which only 21 invoke Filer operations. There are also BCU messages communicated between prime and coupling entity handlers; in SCHM0004 test runs, prime entity handler ENTY0004 sends 59 BCU messages to each of its coupling entity handlers. The number of disc accesses, and the number of Filer/disc-controller record block messages, required during an application run, is more difficult to determine. The P-DB4GL test applications have used a crude filing simulation, and the number of such messages in a fully functional P-DB4GL application depends on the type of filing algorithms and size and number of record buffers used; but it is likely to be of the same order as the number of entity to Filer messages.

The data transfer rates required from the transputer links, needed to support the communication loads produced by the P-DB4GL messages, depends not just on the number and frequency of the messages, but also on the size of the messages. The size of the messages is determined by two things: the amount of data (or control) information contained in the message; and the size of the additional protocol information attached to each message (ie protocol overhead). For a BCU message the protocol information is used to indicate various properties of a message, such as:

- its type (Attribute or Entity action, Request or Reply);
- the source and destination processes;
- the message length.

Appendix B provides detailed descriptions of the message formats and Occam protocols defined for the message channels.

The different types of P-DB4GL messages vary with respect to their proportions of data and protocol overhead content. Each BCU message has a protocol overhead of 12 bytes; for a Request-Reply message pair this doubles to 24 bytes. The data/control information contained in BCU messages used in the P-DB4GL test application is small, typically one to ten bytes. For test data Set 4, the sum of all BCU messages communicated between the User Harness and test schema SCHM0004 amounts to 2286 bytes, of which 2160 bytes is protocol overhead. This gives an overhead:data ratio of approximately 17:1. Therefore, of the data transfer rate needed to support

this communication load, only 5.5% of it is being used to transport database data, the remainder is being used to support the BCU message overheads.

The protocol overhead for attribute-based BCU message is quite considerable, and an aspect requiring further investigation in the fully functional P-DB4GL designs. A possible improvement to the BCU communication overheads is the adoption of a record-based message format. For test data Set 4 on SCHM0004, this would reduce the total communication size to 378 bytes and an overhead:data ratio of 2:1. This would bring BCU communication in line with other types, such as the record-based entity/Filer message channels. To some extent, the BCU protocol:data ratio of the test applications is not representative of genuine P-DB4GL applications. The data content is too small - more realistic values for data sizes are ten times larger, giving a ratio of approximately 2.2:1. However, increasing the data content may improve the ratio, but it also increases the total size of message communication.

Test data Set 4 on SCHM0004 with three coupling entities gives a total message communication for all channels of 6711 bytes (264 BCU request-reply message pairs). The benefit of the concurrent data access algorithms for this test application were obtained with increasingly larger test schemas (ie more coupling entities) executed entirely on one processor - the largest test application had 18 coupling entities. For 18 coupling entities, the total BCU message communication is 28836 bytes. Of this total, only 2286 bytes are communicated on an external link, the remainder is communicated on channels placed in memory. If this application code was distributed across a network of processors and realistic data sizes were used, then the total communication size from all the BCU message channels would be approximately 50 Kbytes. Most of this would have to be transferred over external channels placed on links, and when combined with the additional BCUX and BCUXR protocol overheads of the multiplexing and routing processes, it represents a significant load on the interconnecting links.

Because of the high communication loads, arising from the large numbers of BCU messages with large protocol overheads, and the message latency caused by multiplexing channels and routing messages through a network, the issue of communications balancing is crucial to any designs for a fully functional multiprocessor P-DB4GL implementation. Identification of communication "hot-spots", and the clustering together of processes with

high communication levels, is essential if the performance benefits of the concurrent P-DB4GL processes are to be realised. The particular processes and channels of high communication levels will vary from one application to another, but some general observations can be made. Applications with very few schema links in the data access schema will tend to lower levels of communication between schema and entity handlers. Similarly, applications with smaller numbers of coupling entities, or with coupling entities that are not subject to prime entity updates, will tend to lower levels of communication on the prime to coupling entity handler channels. It is envisaged that significant amounts of experimentation with different software configurations would have to be performed by the system designer, in order to find the best communication balance for each P-DB4GL application. To allow balanced configurations for all of the different P-DB4GL applications that are likely to be executed on a given system, it would also be desirable to have a transputer network that permitted some hardware reconfiguration in between application executions (ie inter-application reconfiguration).

6.4 Designs for P-DB4GL Hardware Configurations

Following the prototype P-DB4GL implementation, and the subsequent investigation of message routing in a multiprocessor version of a fully functional P-DB4GL system, an important design consideration has emerged. The combination of light processing loads and high communication requirements of the P-DB4GL data access processes, indicate that these processes should be distributed across as few processors as possible with communication "hot spot" processes clustered together. Unfortunately, the many filing processors, needed to support a realistically sized fully functional P-DB4GL application, create problems when there are not sufficient data access processors' links to connect them to.

However, the analysis of the communication loads on the various channels connecting different process types shows significant differences in size, frequency, and protocol overhead for the different types of messages. Channels defined for BCU messages have the highest communication loads, whereas channels defined for the communication of record-oriented Filer messages have lower communication loads. This allows the possibility of multiplexing together some of the Filer channels, so that several Filers can be connected to each data access code processor. Figure 6.10 illustrates such a

configuration. Three Filer processes, executing on T400 or T425 processors, are connected to a multiplexor process on a T212 or T222 processor. The multiplexor processor is connected to a data access processor, where the multiplexed Filer channels are then demultiplexed. Each data access code processor can have three Filer multiplexors attached to it; thereby allowing up to nine Filers to be connected to a single data access processor. Typical Filer message sizes are 10 to 1000 bytes. Using a counted array Occam channel of INT16::[]BYTE, and with the overlapped-acknowledge (see Appendix I) on 20 Mbit/s links, an effective data transfer rate at the Filer multiplexor to data access process channels approaching the maximum possible (of approximately 2 Mbyte/s bidirectional) could be attained.

Concerning the network topology of the multiprocessor implementation of P-DB4GL; the ring originally proposed has been found to be unsuitable for larger networks. The ring initially seemed attractive because it required only two of each processors' links for its construction. The remaining two links were available for connection to disc controllers and terminal drivers. If each data access processor is altered for connection to three Filer multiplexor processors (as in Figure 6.10), only one link remains for connection to other P-DB4GL code processors. Therefore, a separate network constructed from simple 16 bit T222 transputers with suitable router processes is used to interconnect the P-DB4GL code processors (Figure 6.11). Network topologies with shorter message distances than rings, such as trees, arrays, and hypercubes, can be composed using the T222 interconnection transputers. Although more complex topologies may reduce message distance, the routing algorithms are generally more complicated, and the additional processing involved may slow down the routing of messages, as in for example [Stringer89] where ring and hypercube topologies are used for routing messages in a database application.

A fully functional P-DB4GL system can be expected, under normal operating conditions, to be used to run several different P-DB4GL applications. Given that each application is likely to have different resource requirements, in terms of numbers of files, discs, and processors, and will probably provide optimum performance only on a particular network topology. Greater flexibility can be obtained by using a reconfigurable crossbar switch for processor interconnection. Figure 6.12 illustrates such a design: the crossbar switch is provided by one, or more (depending on the size of the network), C004 devices [Inmos89b]. P-DB4GL code processors, for data access, user

processes, and host interface code; along with Filer multiplexor and Router processors, are all connected to the link crossbar switch. Before an application is loaded and executed on the transputers, the crossbar switch is programmed to connect the transputer links to the desired network topology. This inter-application network reconfiguration not only provides greater flexibility for executing many different applications, but also facilitates simpler experimental reconfiguration during application development when optimum configurations are sought. It is not envisaged that this reconfigurable design be used for network reconfiguration during application execution (ie intra-application reconfiguration), as this would require considerable redesign of the P-DB4GL application code.

6.5 Code Factoring

In the prototype P-DB4GL system it is the data access processes (schema and entity handlers) that have received the most attention. The data access processes have been fully implemented; whereas the user processes (screen and window processes), needed to complete a database application, have been simulated by the User Harness. However, one of the design considerations relevant to the construction of a fully functional P-DB4GL is an examination of this code factoring used in the prototype P-DB4GL system. The classification into data access and user processes, and the apportioning of functionality amongst the different processes, has been taken directly from the earlier implementations of DB4GL [Ewin85b] [Poole87].

One of the main findings arising from the construction and testing of the prototype P-DB4GL test application is the performance advantage of concurrent data access algorithms over sequential algorithms. These concurrent data access algorithms (and concurrent algorithms for User processes too) might better be represented by decomposing the existing processes into many smaller processes. Reducing the size, and hence increasing the number, of concurrent processes in a program (ie reducing the granularity of concurrency) permits more refined load balancing. Another reason for examining the application code factoring is to see if a closer match between the objects in the application model specification and the processes in the concurrent Occam implementation is possible (or desirable).

The schema handler process in particular merits further investigation with respect to the code factoring of the P-DB4GL applications. Currently, the schema handler process is used in the implementation of an application task's data access schema. The main function performed by the schema handler is prefetching from the filestore of related records (ie realisation of schema links). An alternative means of providing this function is to refactor the schema handler into separate Information Unit Group (IUG) processes (Figure 6.13). The schema handler process currently represents a point of high communication load - all messages from user processes to entity handlers must pass through it. The creation of separate IUG processes would eliminate this potential communication bottleneck.

A more radical refactoring of application code worth investigating is the creation of Presentation Object (PO) processes (Figure 6.14). Each PO process would be decomposed into separate processes representing both: the data access functions (IUG processes); and the functions currently provided by user processes (Presentation Unit and Process Task processes). This represents a very direct translation from application task specification to concurrent Occam process implementation. Factors likely to affect the feasibility of this direct translation are: re-usability of the constituent processes for application building; granularity of concurrency for efficient load balancing; and patterns of inter-process communication affecting the communication loads (ie frequency of messages, size of protocol overheads, number of channels needed and complexity of interconnection).

6.6 Code Generation

The methods of code generation and application loading and initialisation for P-DB4GL applications have changed from those used in the original DB4GL. In DB4GL, code generation tools take skeleton COBOL programs, for code modules such as an entity and schema handlers, and by the addition of extra lines to this source code produce completed programs ready for compilation. Following compilation, the loading and initialisation of the generated application on the single-tasking single processor PC is quite straightforward and handled by the operating system and COBOL environment. In the prototype P-DB4GL system the method of code generation is similar, except that it is done by hand, not automatically by a generator tool. Skeleton Occam procedures for the data access processes are completed by the addition of lines of value declarations; following

compilation, the completed procedures are used to construct test applications, which are then configured and loaded onto the transputer networks by tools in the development environment.

In the course of designing a fully functional multiprocessor version of P-DB4GL, and implementing and testing some of its features, the method of code generation has been modified. The data access processes have been parameterised; so that, a generic compiled entity handler (or schema handler) Occam procedure can be instantiated to a specific data access process (Figure 6.15). This obviates the need for additional compilations each time a new data access process is required. However, a large number of parameter values is needed to instantiate a data access process, and it would probably be better to replace the parameters with an initialisation channel (Figure 6.15).

These changes to the method of code generation affect the role of the P-DB4GL data dictionary in respect of application loading and initialisation. If, as in the prototype P-DB4GL system, additional compilations are needed for each instantiation of new application code processes; then the data dictionary node must load applications by downloading compiled code into the network. But when new code processes can be instantiated without further compilations, it becomes possible for the data dictionary node to load and initialise an application by simply downloading a description of the required application to the Kernel processes at each node in the network. The Kernel processes can then invoke instances of appropriate component processes of the required application at their nodes. This permits a more interpreted role for the data dictionary, and enhances the rapid application generation and prototyping features of the P-DB4GL system.

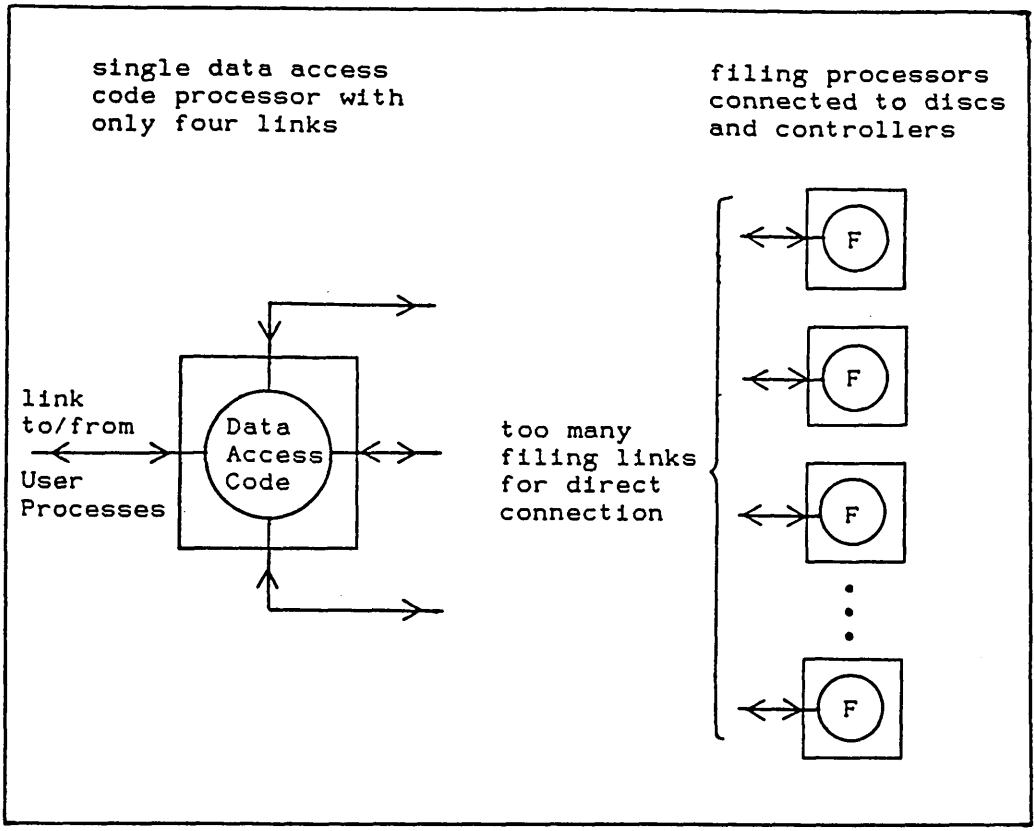


Figure 6.1 - Insufficient Links for Filer Connection

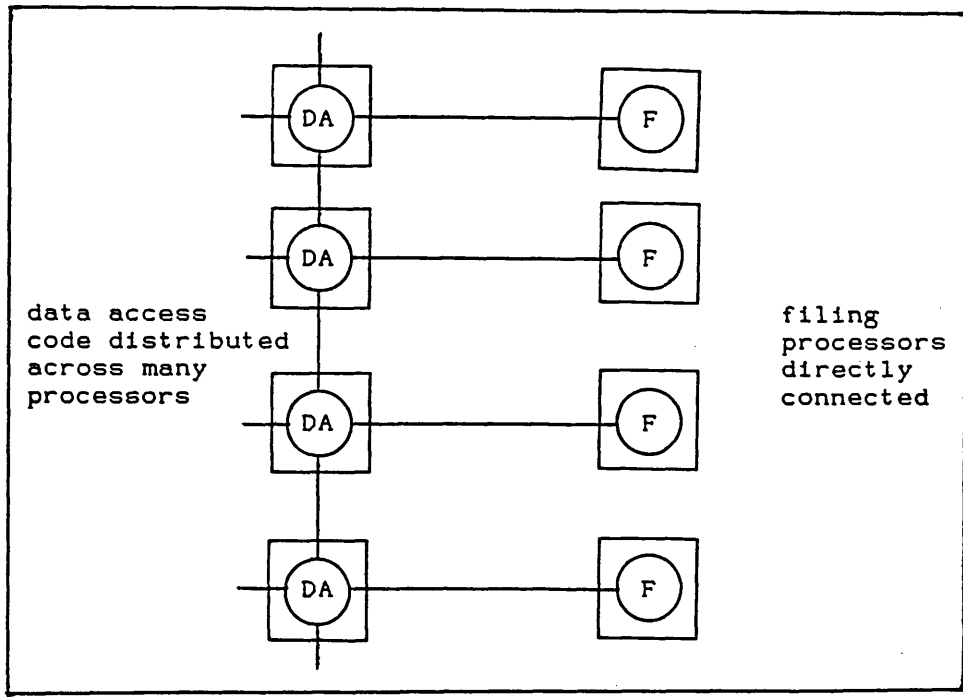


Figure 6.2 - Distributing the Data Access Code

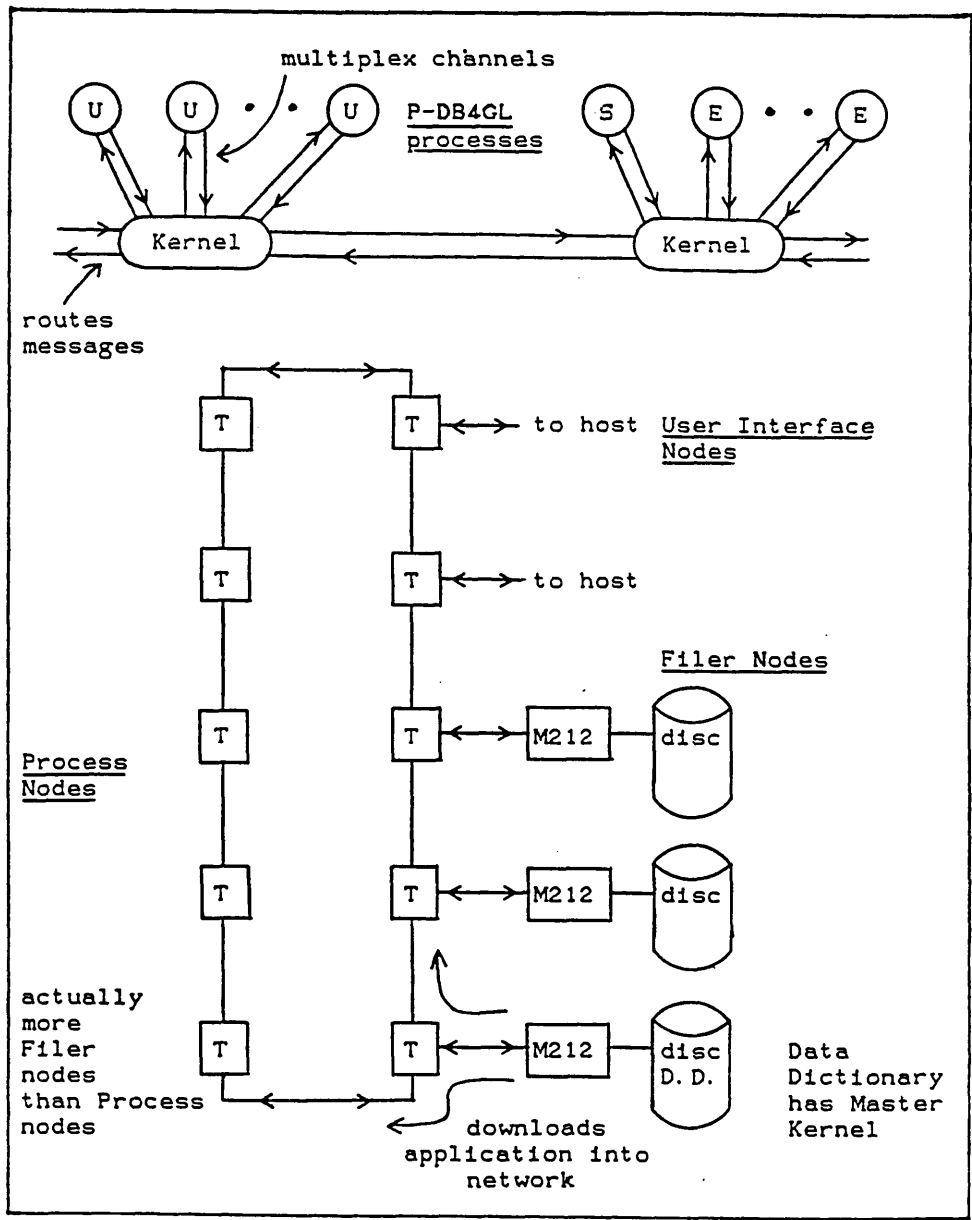


Figure 6.3 - Designs for a Fully Functional Multiprocessor P-DB4GL

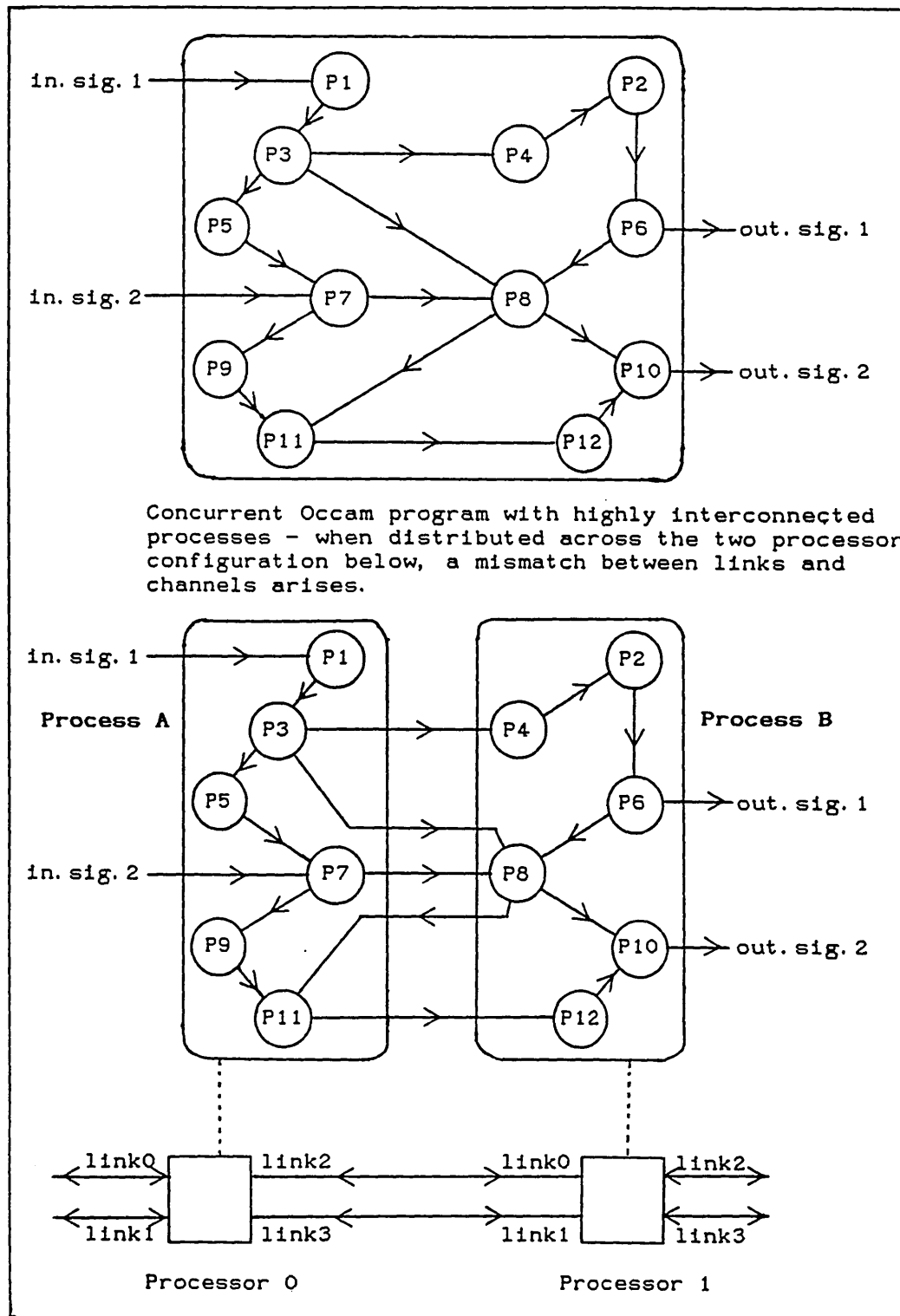


Figure 6.4 - Configuration Mismatch Between Channels and Links

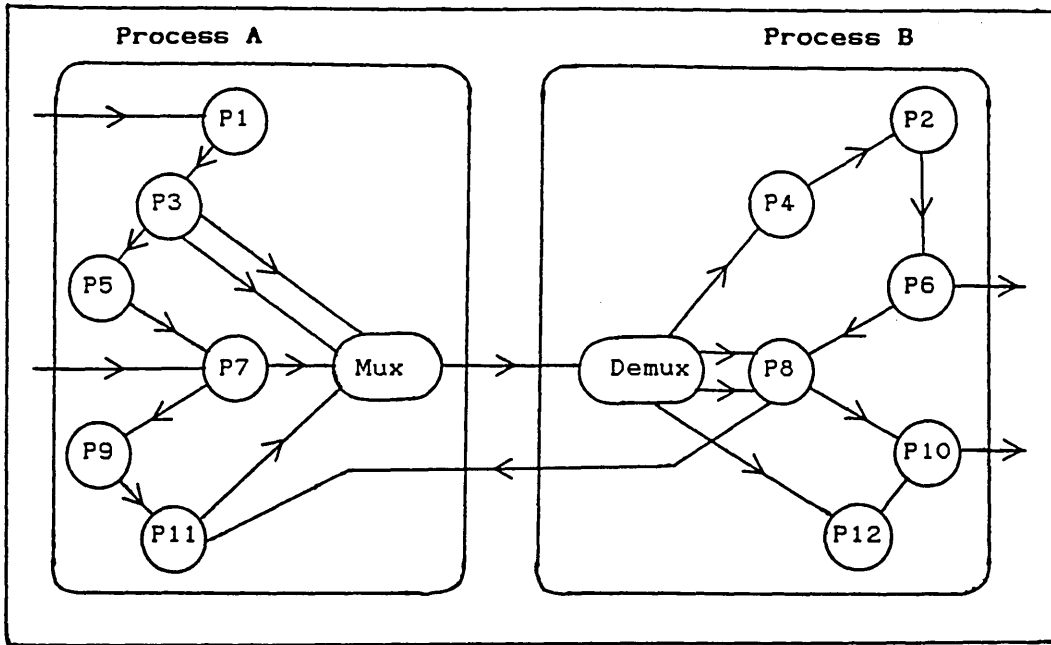


Figure 6.5 - Addition of Multiplexor Processes

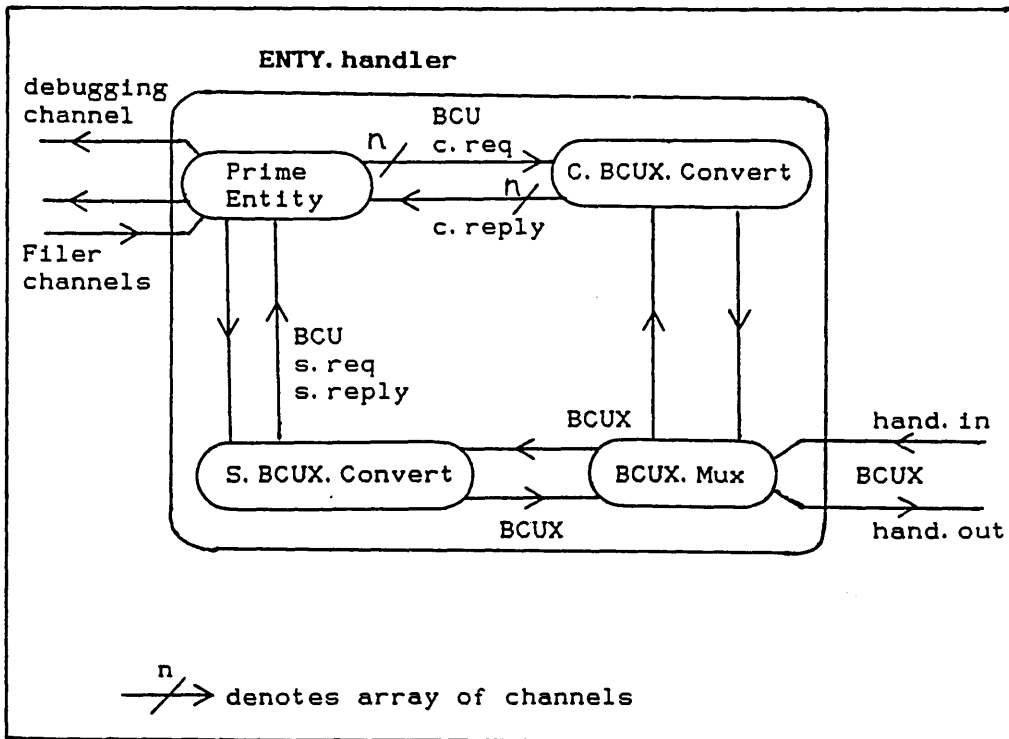


Figure 6.6 - Modified Prime Entity Handler

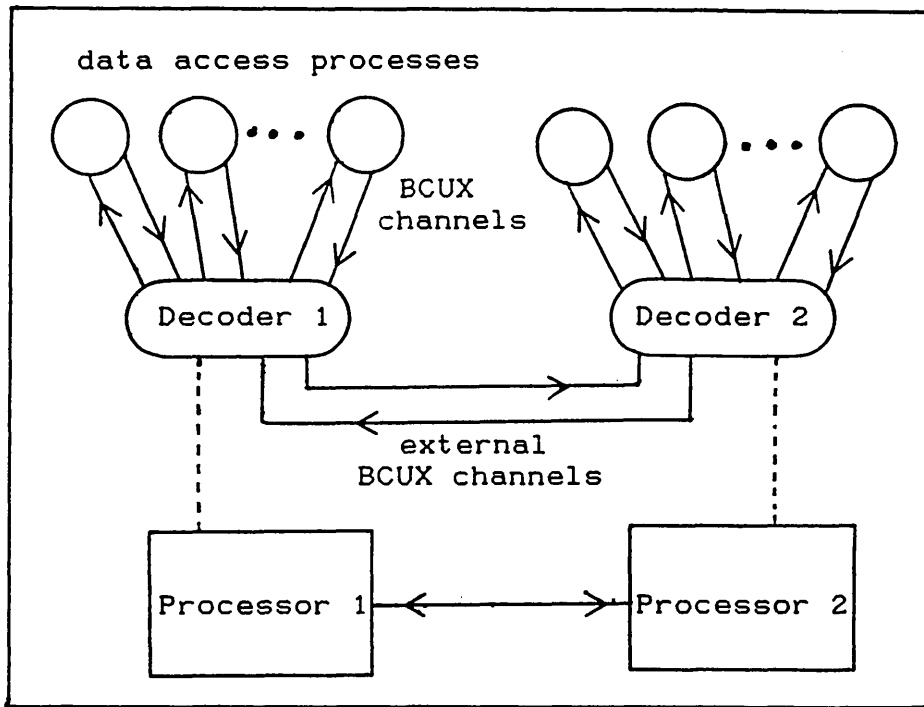


Figure 6.7 - P-DB4GL Decoder Process

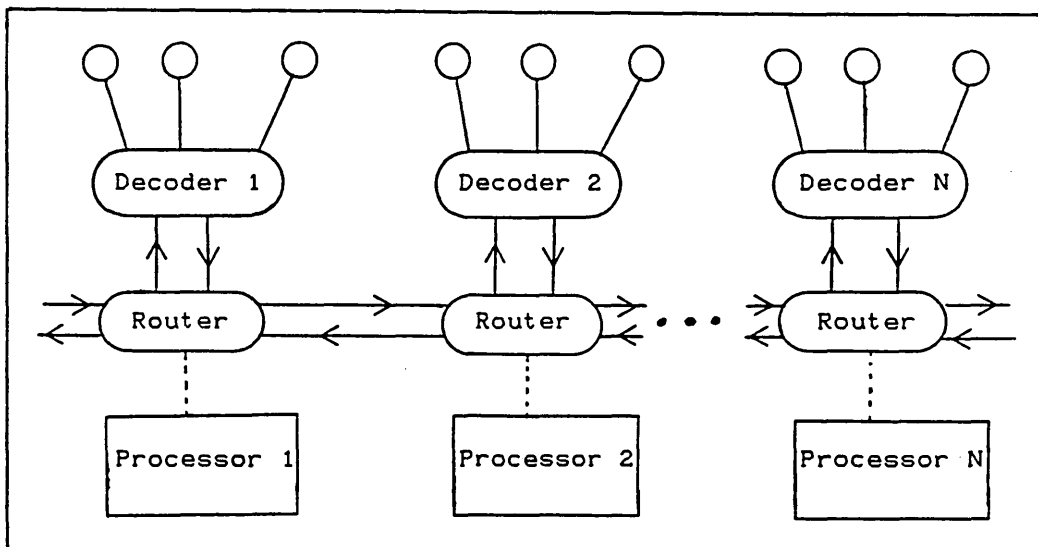


Figure 6.8 - Routing Messages Through a P-DB4GL Network

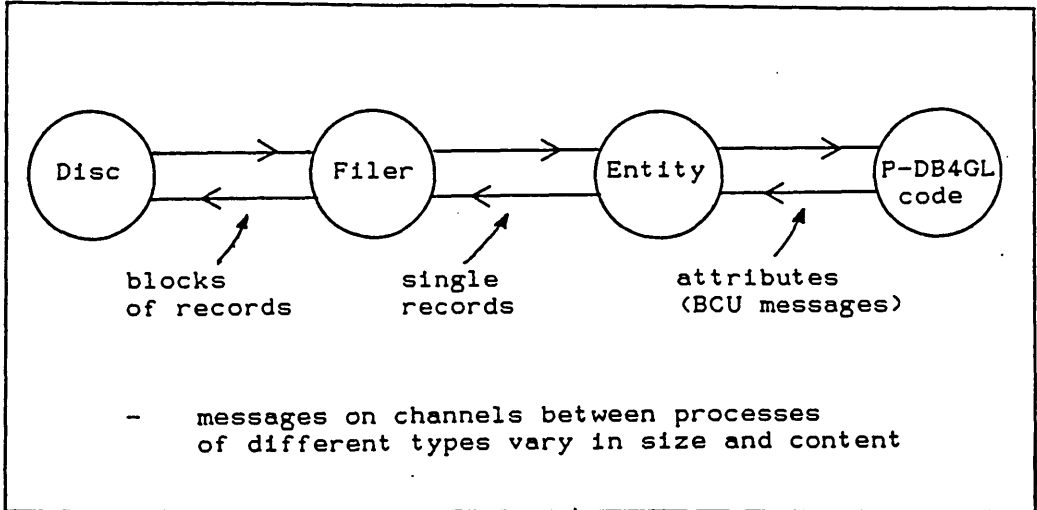


Figure 6.9 - Different Types of P-DB4GL Messages

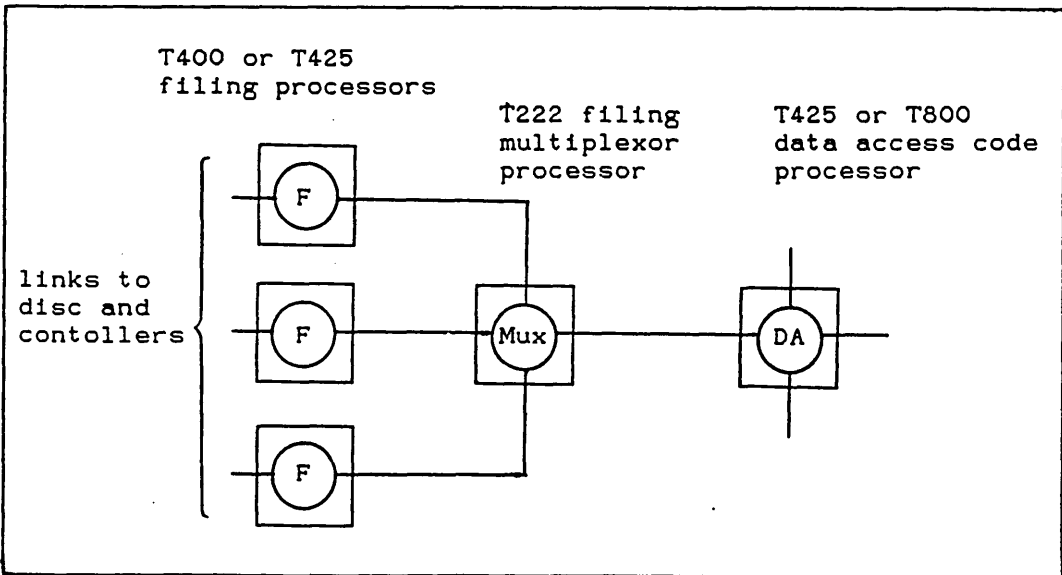


Figure 6.10 - Multiplexing Filer Channels

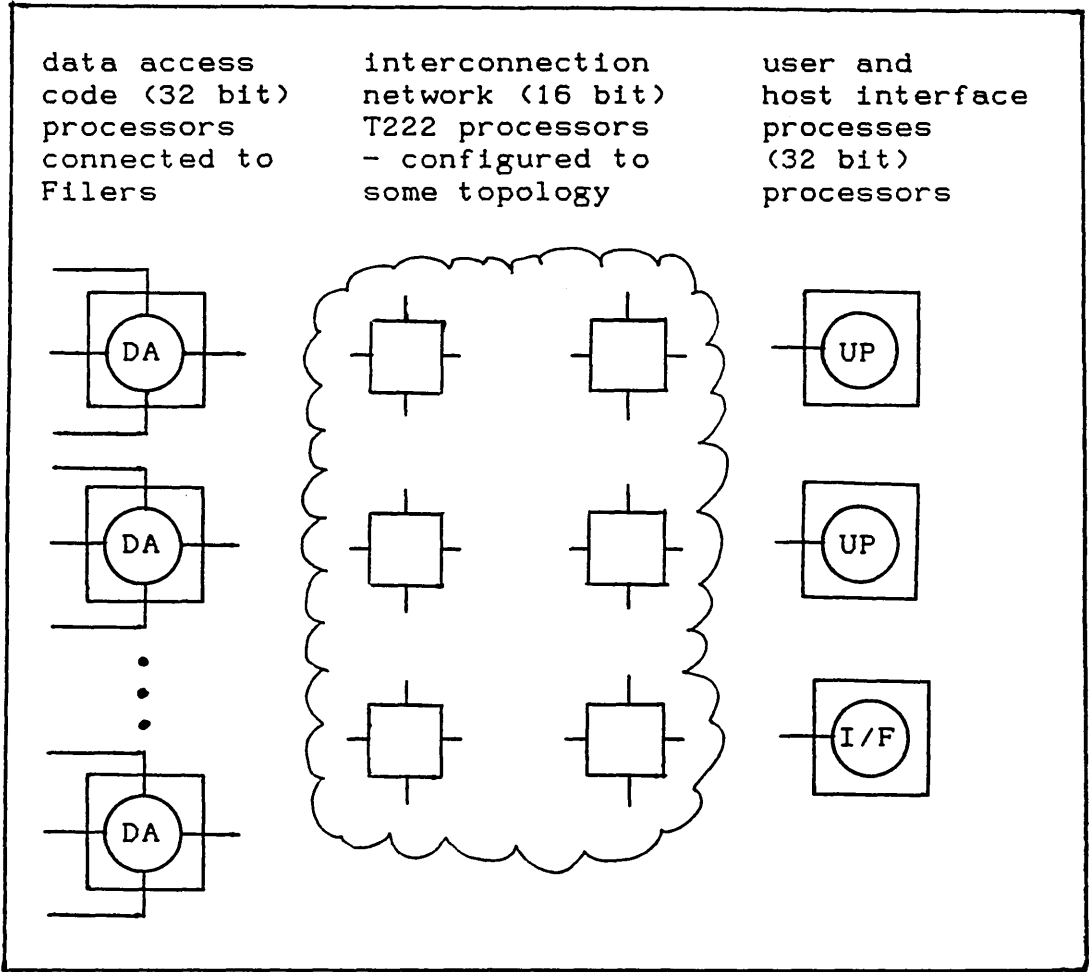


Figure 6.11 - A Separate Interconnection Network

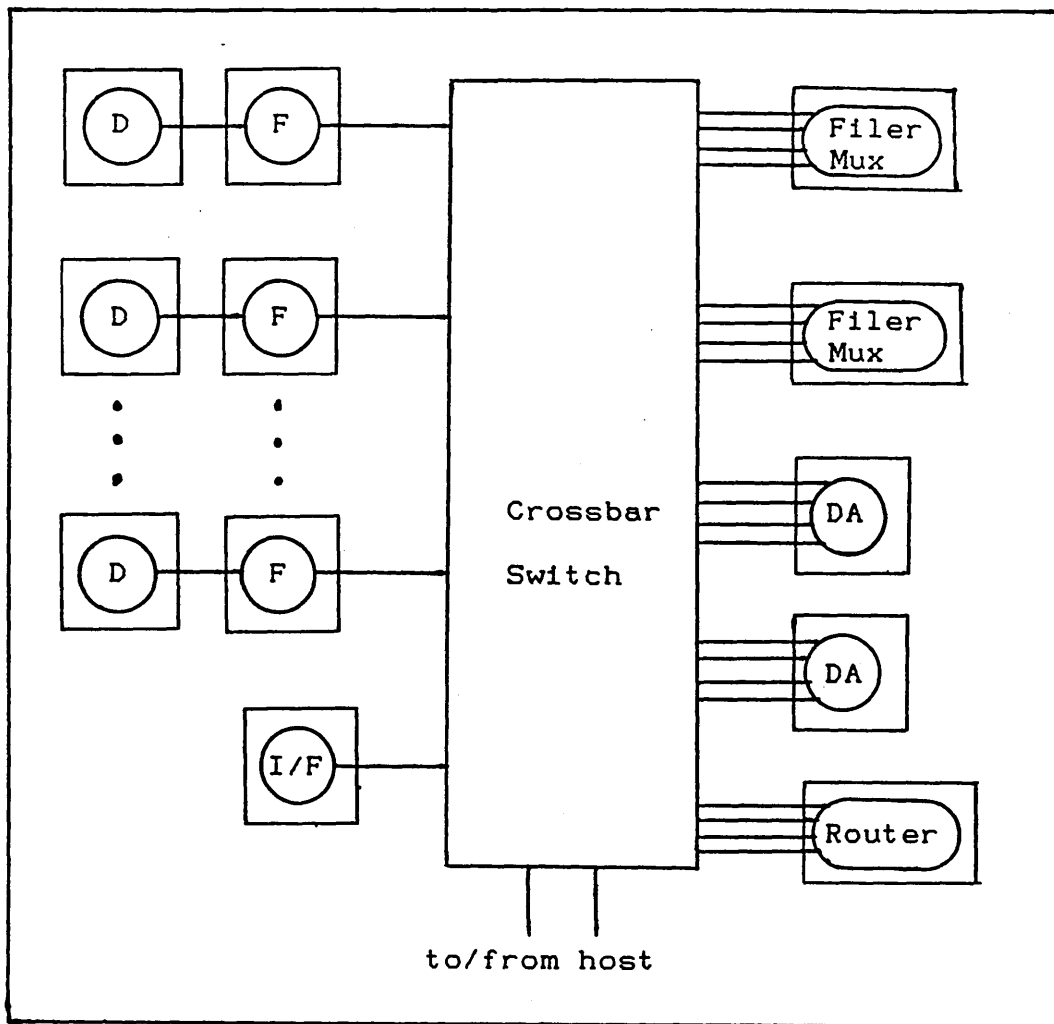


Figure 6.12 - A Reconfigurable Network

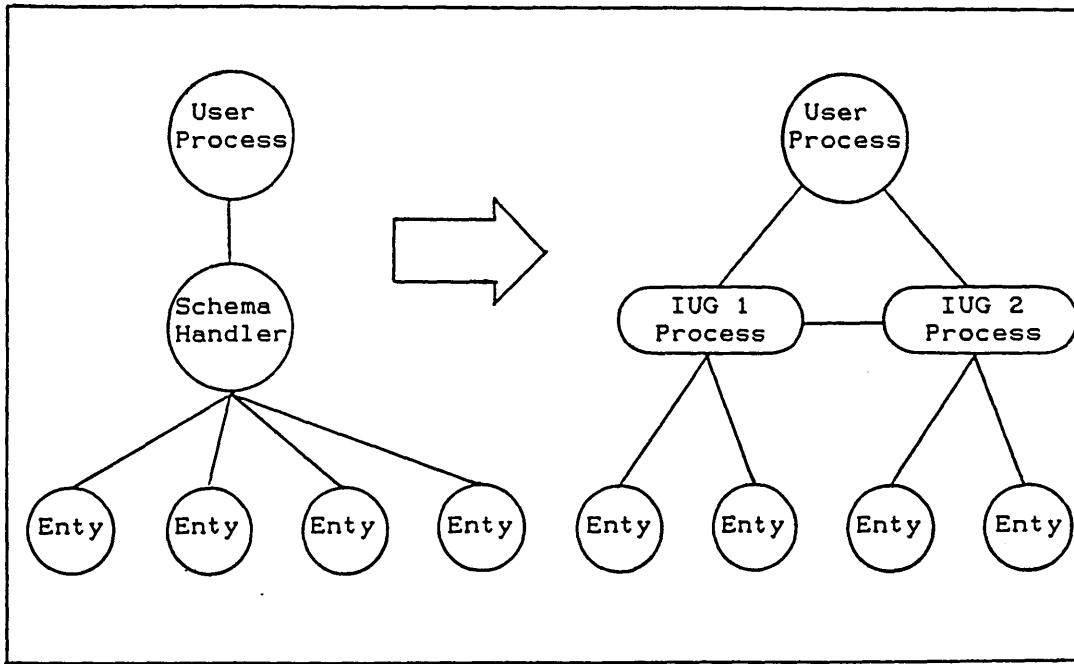


Figure 6.13 - Replacement of Schema Handler by IUG Process

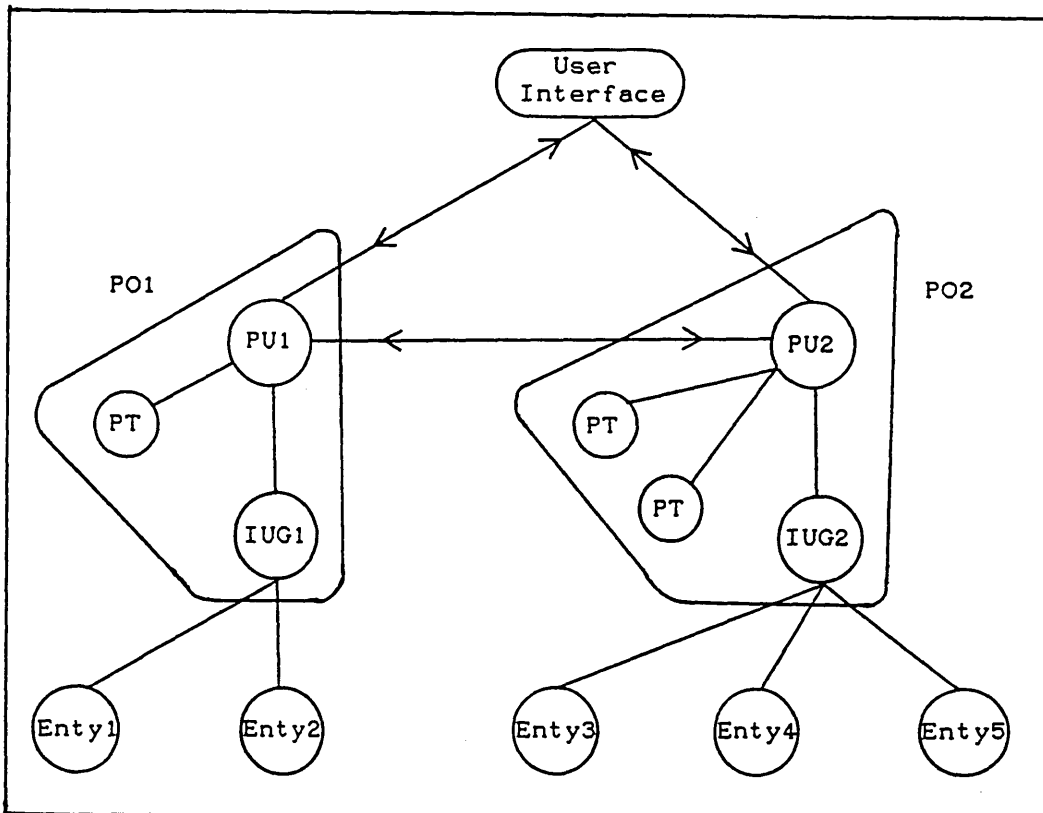


Figure 6.14 - An Alternative Presentation Object Implementation

i) Insertion of constant declarations into a procedure skeleton

```
PROC SchmXXXX(...)
```

```
VAL []BYTE schm.id IS..
VAL INT no.of.entites IS..
VAL [][]BYTE schm.links IS..
```

```
-- main body
SEQ
... schema handler code
:
```

ii) Parameterisation of generic handler modules

```
PROC Schma.handler(VAL []BYTE schm.id IS..
                  VAL INT no.of.entites IS..
                  VAL [][]BYTE schm.links IS..
                  ... channels parameters
                  )
```

```
-- main body
SEQ
... generic schema handler code
:
```

iii) Initialisation message

```
PROC Schm.handler(CHAN OF INIT.MESS init.chan,
                  ... other channel parameters
                  )
```

```
-- main body
SEQ
... initialisation routine
... generic schema handler code
:
```

Figure 6.15 - Methods of Code Generation

Chapter 7

P-DB4GL Project Evaluation

7.1 Summary

The DB4GL application model, in which database applications are defined as a set of concurrently executable message passing Presentation Objects, was found to be highly suitable for application specification and generation. However, the implementation of the generated applications as sequential program structures on a single-tasking single processor microcomputer, involved design transformations (mentioned in chapter 2 section 2.5, and chapter 5 sections 5.2,5.4) that obscured the potential for parallel execution inherent in the application model.

In the Parallel-DB4GL (P-DB4GL) project, an attempt has been made to retain the parallelism, inherent in the specification of DB4GL database applications, by using a concurrent programming language to implement the applications. The CSP [Hoare78] derived Occam language has been used for the implementation, rather than an object-oriented programming language. The constituent message passing Presentation Objects of a DB4GL database application are realised as communicating processes in a concurrent Occam program. This direct translation, from objects to processes, was relatively straightforward, involving very little in the way of design transformations, and the resultant Occam programs have been mapped onto transputer networks for parallel execution.

The development of a prototype P-DB4GL system has tested the feasibility of this approach, and led to the design of a fully functional P-DB4GL. The design, and partial implementation, of the fully functional P-DB4GL has indicated many opportunities for further research, both within the P-DB4GL project and for parallel implementations of object-oriented systems. These research opportunities are described in the following sections of this chapter.

7.2 Communication Loads and Object Clustering

The processing loads of P-DB4GL processes are very light, but the communications overheads are substantial. This results in a high communication/processing ratio which is not matched by the transputer based parallel hardware. Load balancing was not a major concern, because

the P-DB4GL applications do not significantly load the processors, but communications balancing was a serious problem. The communication patterns within the P-DB4GL applications are uneven; "hot-spots" of intense message passing and their consequent high inter-process communication loads are present. It is important to identify these communication "hot-spots" and place the participating processes physically close together on the same processor to prevent applications from becoming communications bound. This "clustering" together of objects with high levels of inter-object message passing (and the resultant high communication load on the message channels), is something that should be considered by any designer implementing object-oriented systems on distributed memory hardware such as transputer networks.

7.3 Object Inheritance

An object-oriented feature missing both from the DB4GL application model and from the P-DB4GL implementation is inheritance. The application model contains little in the way of a generalization/specialization hierarchy of classes, although some work has been done to support the concept of subclasses of data type domains [Hird89]. Within the P-DB4GL system there is no support for late binding and inheritance of methods at run-time. However, in the context of the DB4GL prototyping application generation cycle, there is the possibility of extending the data dictionary data model to support classes of object types, for example, the classes of prime and coupling entity handlers as subclasses of the entity handler object type. At application generation time, when instances of the object types are created, some form of method inheritance from superclass to subclass object type might be provided, such as, the inclusion of generic entity handler integrity rule processing methods inside an instance of a generated coupling entity handler object.

7.4 Occam 2 as an Implementation Language

The Occam language provides encapsulation and information hiding within processes. It supports synchronous message passing using well defined protocols on point-to-point inter-process channels. However, the language is rather minimal: it lacks some dynamic properties such as recursion; process creation and resource allocation is static; complex (and time consuming) user defined processes need to be written in order to support

communication patterns other than synchronous point-to-point. As a consequence of these language properties, the design, implementation and testing of Occam programmed systems can be tedious and lengthy. The development of good software tools and the inclusion of some language extensions to Occam are needed.

The field of object-oriented design is becoming more established [Wirfs-Brock90] [Henderson-Sellers90]; and increasingly examples of object-oriented design applied to concurrent and real time application areas are appearing [Plessman88] [Heever89] [Agha90]. In the development of Parallel-DB4GL, concurrent and real-time concerns have been important issues. Furthermore, the principles of object-oriented design have influenced the implementation of the prototype P-DB4GL system. An area of further research, likely to prove beneficial to projects similar to P-DB4GL, is an investigation of closer integration of concurrent programming languages, such as Occam, into established object-oriented design and structured design methodologies, for example Occam and Mascot3 [Knowles90].

7.5 Improvements in Secondary Storage Technology

The designs for a fully functional P-DB4GL system presented in Chapter 6 are not tied to any particular secondary storage (disk) technology. Although the initial P-DB4GL design (Chapter 5) included M212 disc controller transputers, and the presence of these devices has been assumed in the fully functional P-DB4GL designs, they are not essential to the P-DB4GL multiprocessor architecture. The M212 device is no longer manufactured by Inmos, and has been replaced by a T222 based SCSI interface board [Inmos89c]. This new SCSI (Small Computer System Interface) disc interface board can be substituted directly for the disc processes and controllers in the P-DB4GL design (although with appropriate changes to the software interface at the Filer channels).

In order that disc access times and data transfer rates can keep up with the increasing speeds of CPUs and main memory, a disc technology known as RAID (Redundant Arrays of Inexpensive Disks) has recently been developed [Patterson88] [Patterson89] [Ng89]. The principle behind RAID is the replacement of a single large expensive disc with many smaller less expensive discs. The goals are improved reliability, through redundancy, and improved transfer rate, via techniques such as disk striping [Salem86].

In the P-DB4GL designs, only a single disc and controller has been assumed at each Filer node. However, this is not fundamental to the P-DB4GL architecture, and some Filer nodes could be modified with the addition of a RAID subsystem to provide either increased transfer rate, or improved reliability, if needed. Similarly, with other storage technologies such as optical and opto-magnetic storage [Bate89] [Williams89], some Filer nodes could be altered to store their data on these devices, rather than the small Winchester disc drives proposed in the original P-DB4GL designs. Some Filer nodes might even use semiconductor-based secondary store; but for reasons of size, cost, volatility, and security, it is likely that most of the data in typical P-DB4GL systems would be stored on magnetic (disk) secondary store.

7.6 The Next Generation of Transputer Products

The next generation of transputer products announced by Inmos, which includes the H1 microprocessor [Inmos90] and C104 router device [May90], opens up new possibilities for a transputer based Parallel-DB4GL. The H1 processor has claimed peak performance in excess of 150 MIPS and 20 MFLOPS, and a total communications bandwidth of 80 Mbyte/s. The H1 uses a new link technology with packet based communication, and provides a "virtual channels" facility for inter-process channel communication when programmed in Occam. When combined with the C104 router device, which provides hardware routing of the communication packets, the potential performance for message passing distributed architectures is greatly improved.

A problem with the implementation of P-DB4GL on the current T-range of transputer products is the very high levels of communication, combined with light processing loads, encountered in P-DB4GL applications. Although the T-range microprocessors provide approximately 2 Mbyte/s maximum bidirectional data transfer rate on each link (8 Mbyte/s total bandwidth for a four link device), designing P-DB4GL code that can effectively use this communications bandwidth is very difficult. When combined with the software channel multiplexors and message routing processes, needed in the distributed versions of P-DB4GL application, the effective usable communications bandwidth was severely reduced to perhaps 10% of the theoretical maximum (see Appendix J). This reduced communications bandwidth, and the message delays associated with routing

through a network, impaired the attainable performance of distributed P-DB4GL applications.

However, the greatly improved communication bandwidth of the H1, and the hardware support for virtual channels and message routing, should provide sufficient message passing capacity for high communication programs such as P-DB4GL database applications. In fact, the massive processing power (in terms of MIPS) of a single H1 processor may be sufficient to support an entire P-DB4GL application. A possible design for an H1 based P-DB4GL architecture is shown in Figure 7.1: a few, perhaps only one, H1 processors are connected to a C104 router, this is connected via link adapters to T-range microprocessors such as T400, T425, or T800 used as filing processors, and these are connected to discs and controllers. This is only an outline of a possible H1 based design. As full technical disclosures are made available, and actual devices are manufactured, more detailed designs can be considered.

To accompany the new range of transputer products, a new version of Occam is planned [Barrett90]. This new version extends the language with many features found in other high languages, such as, records, user defined data types, and modules. New shared object types, such as buses of channels, with synchronization mechanisms additional to the basic Occam communication primitives (? and !) are also planned. The new version of Occam has an improved configuration language that separates hardware description from software configuration; in particular, the current requirement for the programmer to place channels at specific links is removed.

In the course of implementing P-DB4GL, the Occam 2 language has been found to be rather minimal in respect of data structure manipulation, and the incorporation of record structures into Occam is welcomed. The changes to the configuration language should make application development much easier, especially during experimental phases when many different configurations are tested for processing and communications load balancing. This experimental configuration phase might possibly be automated by an appropriate P-DB4GL tool. Furthermore, the H1 processor contains some instruction level support for implementing operating system kernel functions. It is therefore likely that many more operating systems will be either developed for, or ported to, the new H-range of processors. When

such operating systems arrive, a useful direction for further research is an investigation of possible support for P-DB4GL applications from these operating systems.

7.7 Advantages of a Prototyping Development Cycle

There are many problems associated with performance improvements in database systems using parallel processing hardware. Problems such as, load balancing, communications balancing, data partition and data skew, choice of network topology, distribution versus replication, message passing mechanisms, buffering, etc.. These problems are most acute in enquiry processing systems supporting ad hoc queries, in which, data sizes, communication patterns, and processing requirements of individual queries are not known in advance, and must be determined at run-time in order to achieve efficient parallel resource utilization.

However, P-DB4GL database applications do not suffer from these problems to the same degree for the following reasons. P-DB4GL applications are transaction oriented, and the access paths through the database are known in advance. Also, in the course of the application development cycle, the processing loads and communication patterns of particular applications can be gauged. It is therefore possible to experiment with the system configuration, adjusting the topology and data partitioning at each iteration in the application development cycle; thus, efficient solutions for each application can be found. This parallel resource experimentation would be an additional task performed by the system designer during the development cycle, although the use of high level P-DB4GL tools (similar to the existing tools used in the functional/interface specification of DB4GL applications) would relieve the system designer of the burden of low level parallel implementation details.

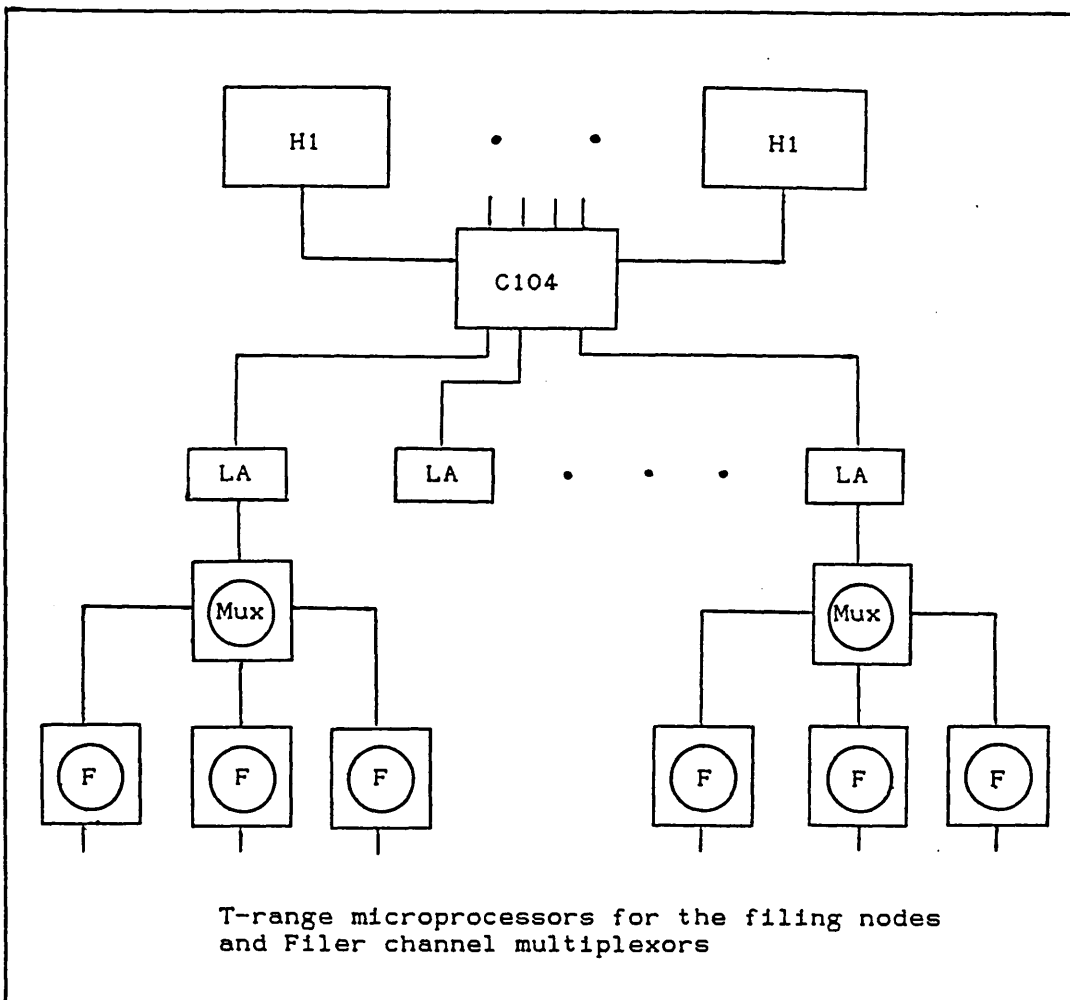


Figure 7.1 - An H1 Based P-DB4GL Design

Chapter 8

Conclusions

The development of the Parallel-DB4GL (P-DB4GL) system has demonstrated the benefits of using a concurrent language to implement the DB4GL database applications. The sequential COBOL implementation of DB4GL, on a single-tasking single processor PC, suffered from disc I/O bottlenecks that seriously impaired the performance of both the application generation tools and the generated applications. The concurrent P-DB4GL implementation was able to take advantage of the parallel processing benefits provided by a transputer-based multiprocessor architecture. The transputer's micro-coded low level scheduler provides efficient support for concurrency on a single processor. The transputer's special communication facilities permit easy connection of multiple processors, including multiple disc I/O subsystems. The high disc I/O bandwidth of the multiple disc subsystems in the parallel architecture was used to alleviate the disc I/O bottleneck that had impaired the performance of the original sequential DB4GL implementation. Additionally, the specification of DB4GL database applications as sets of concurrent message passing Presentation Objects, facilitated a relatively simple translation to an implementation in Occam of concurrent communicating processes.

The P-DB4GL project has demonstrated the suitability of distributed memory message passing architectures for database, and data intensive, application areas. Given the many disc and other peripheral I/O processors needed to support the high levels of data throughput typically found in such applications; it is not clear that parallel architectures based on shared memory or shared buses could cope with the high data throughput demands, particularly as applications are scaled-up to greater sizes. The P-DB4GL applications generate large numbers of inter-process messages, which cause a high communication demand on the transputers interconnecting links. The effective data transfer rates available for the transputers' links in the T-range of microprocessors was found to be barely sufficient for such high message passing loads. However, the introduction of the new H-range of transputers promises vastly improved communication bandwidths needed by these applications with very high message passing loads.

The P-DB4GL database applications have been designed for direct execution on transputer networks without an intervening operating system.

Operating systems for transputer networks are now becoming available; but it is difficult to assess the effects, in terms of overheads and transparency, that such operating systems might have on P-DB4GL applications. The operating systems on conventional single processor machines are often bypassed in a database application by the DBMS, sometimes the hardware itself is supplemented with a special database machine. It is likely to be the case with database applications executing on parallel architectures too. The parallel operating system may be bypassed and the underlying hardware made visible to the database application, in order that maximum performance be extracted from the secondary storage (disc) subsystems.

Bibliography

- [Agha90] G Agha
'Concurrent Object-Oriented Programming' Comms. of the ACM Vol 33 No 9 September (1990) 125-141
- [Appleton83] D S Appleton
'Data Driven Prototyping' Datamation November (1983) 259-268
- [Askew88] C Askew (ed)
Occam and the Transputer - Research and Applications
Proc. of the 9th occam User Group Technical Meeting (OUG-9) 19-21 September 1988, Southampton, UK, IOS Press (1988)
- [Atkin89] P Atkin
'Performance maximisation' Inmos Technical Note 17 (72 TCH 017) reprinted in [Inmos89a] 228-246
- [Babb79] E Babb
'Implementing a relational database by means of special hardware' ACM Trans. on Database Systems Vol 4 No 1 March (1979) 1-29
- [Bakkers89] A Bakkers (ed)
Applying Transputer Based Parallel Machines Proc. of 10th Occam User Group Technical Meeting (OUG-10) 3-5 April 1989, Enschede, Netherlands, IOS, Amsterdam (1989)
- [Bal85] H S Bal
'Report Program Generator' BSc Project Report, Dept of Computer Studies, Sheffield City Polytechnic (1985)
- [Baroody81] A J Baroody and D J DeWitt
'An Object-Oriented Approach to Database System Implementation' ACM Trans. on Database Systems Vol 6 No 4 December (1981) 576-601
- [Barrett90] G Barrett
'The Development of occam: types, sharing and modules' in Proc. of the 13th Occam Users Group Technical Meeting 18-20 September (1990) York, UK.
- [Bate89] G Bate
'Alternative Storage Technologies' in [Compcn89] 151-157
- [Beazley84] R C Beazley

'A Data Dictionary Interrogation System' BSc Project Report, Dept of Computer Studies, Sheffield City Polytechnic (1984)

- [Bell86] C G Bell
'RISC: Back to the Future?' Datamation June 1 (1986) 96-107
- [Boehm76] B W Boehm
'Software Engineering' IEEE Trans. on Computers Vol C-25 No 12 December (1976) 1226-1241
- [Booch86] G Booch
'Object-Oriented Development' IEEE Trans. on Software Engineering Vol SE-12 No 2 February (1986) 211-221
- [Briat89] J Briat et al
'PARX: A Parallel Operating System for Transputer Based Machines' in Proc. of OUG-10 [Bakkers89] 114-142
- [Britten80] J N G Britten
'Design for a changing environment' The Computer Journal Vol 23 No 1 February (1980) 13-19
- [BSI88] British Standards Institute
BS 6964 : 1988 Database Language SQL B.S.I. (1988)
- [Cameron86] J R Cameron
'An overview of JSD' IEEE Trans. on Software Engineering Vol SE-12 No 2 February (1986) 222-240
- [Chen76] P P-S Chen
'The Entity-Relationship Model - Toward a Unified View of Data' ACM Trans. on Database Systems Vol 1 No 1 March (1976) 9-36
- [Cobb85] R H Cobb
'In praise of 4GLs' Datamation Vol 31 No 14 July (1985) 90-96
- [Compccon89] COMPCON Spring 89 34th IEEE Computer Society International Conference 27 February - 3 March 1989 San Fransisco, USA, IEEE Computer Society Press (1989)
- [Cook86] S Cook
'Languages and object-oriented programming' Software Engineering Journal Vol 1 No 2 March (1986) 73-80
- [Cooper83] D C Cooper

'A Data Dictionary System' BSc Project Report, Dept. of Computer Studies, Sheffield City Polytechnic (1983)

- [Cox84] B J Cox
'Message/Object Programming: An Evolutionary Change in Programming Technology' IEEE Software Vol 1 No 1 January (1984) 50-61
- [Dearnley83] P A Dearnley and P J Mayhew
'In favour of system prototypes and their integration into the systems development cycle' The Computer Journal Vol 26 No 1 (1983) 36-42
- [DeWitt79] D J DeWitt
'DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems' IEEE Trans. on Computers Vol C-28 No 6 June (1979) 395-406
- [Ewin84] N A Ewin
'Computer Estimating for a Speculative House Builder Using 4th Generation Software Tools' Internal Research Paper R/D/84/3, Dept of Building, Sheffield City Polytechnic (1984)
- [Ewin85a] N A Ewin
'Advanced Application Software for Speculative Housing Companies' MPhil Thesis (CNAAs), Sheffield City Polytechnic (1985)
- [Ewin85b] N A Ewin, F Poole, R Oxley
'DB4GL: A Fourth Generation System Prototyping Tool' Sheffield City Polytechnic Report, Dept. of Building Internal Research Paper R/D/85/2 (1985)
- [Fishman87] D H Fishman et al
'Iris: An object-oriented database management system' ACM Trans. on Office Automation Systems Vol 5 No 1 January (1987) 48-69
- [Flynn72] M J Flynn
'Some computer organizations and their effectiveness' IEEE Transactions Vol C21 (1972) 948-960
- [Forage85] G Forage
'Fourth-generation languages and advanced software development aids' Data Processing Vol 27 No 9 November (1985) 6-8
- [Gallizi90] E Gallizi, M Cannataro, G Spezzano, D Talia

- 'A Deadlock-Free Communication System for a Transputer Network' in Proc. of OUG-12 [Turner90] 11-21
- [Gardarin83] G Gardarin et al
'SABRE: A relational database system for a multiprocessor machine' in [Hsiao83] 19-35
- [Gimarc87] C E Gimarc and V M Milutinovic
'A survey of RISC processors and computers of the mid-1980s' IEEE Computer September (1987) 59-69
- [Gray90a] J P Gray and F Poole
'Parallel-DB4GL: An Implementation of a Self-Describing Object-Oriented Database Application Generator on Transputer Hardware' in Proc. of OUG12 [Turner90] 34-49
- [Gray90b] J P Gray and F Poole
'A Transputer Based Implementation of a Parallel Database System' in A Brown and P Hitchcock (eds) BNCOD-8 Proc. of 8th British National Conference on Databases 9-11 July 1990, York, UK, Pitman Publishing (1990) 32-63
- [Gjessing88] S Gjessing and K Nygaard (eds)
Proc. of ECOOP'88 European Conference on Object-Oriented Programming Oslo, Norway, August 15-17 1988, Springer-Verlag (1988)
- [Grimsdale89] C H R Grimsdale
'Distributed operating system for transputers' Microprocessors and Microsystems Vol 13 No 2 March (1989) 79-87
- [Heever89] R J van den Heever and D G Kourie
'Design of Distributed Systems: Object-Oriented Event - Driven Approach' in [Neishlos89] 113-122
- [Henderson-Sellers90] B Henderson-Sellers and J M Edwards
'The Object-Oriented Systems Life Cycle' Comms. of the ACM Vol 33 No 9 September (1990) 142-159
- [Hird89] B T Hird
'Process Logic for an Expert Database System' PhD Thesis (CNAAs), Sheffield City Polytechnic (1989)
- [Hoare78] C A R Hoare

'Communicating sequential processes' Comms. of the ACM Vol 21 No 8 August (1978) 666-677

- [Horowitz85] Horowitz and Ellis et al
'A Survey of Application Generators' IEEE Software Vol 2 No 1 January (1985) 40-54
- [Howe83] D R Howe
Data Analysis for Date Base Design Edward Arnold Publishers Ltd (1983)
- [Hsiao83] D K Hsiao (ed)
Advanced Database Machine Architectures Prentice-Hall (1983)
- [Hudson89] S E Hudson and R King
'Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System'
ACM Trans. on Database Systems Vol 14 No 3 September (1989) 291-321
- [Hull89] M E C Hull, A Zarea-Aliabadi, and D A Guthrie
'Object-oriented design, Jackson system development (JSD) specifications and concurrency' Software Engineering Journal March (1989) 79-86
- [Inmos88a] Inmos Ltd
Transputer Reference Manual Prentice Hall (1988)
- [Inmos88b] Inmos Ltd
occam 2 Reference Manual Prentice Hall (1988)
- [Inmos88c] Inmos Ltd
Transputer Development System Prentice Hall (1988)
- [Inmos88d] Inmos Ltd
Transputer Instruction Set; A Compiler Writer's Guide Prentice Hall (1988)
- [Inmos89a] Inmos Ltd
Transputer Technical Notes Prentice Hall (1989)
- [Inmos89b] Inmos Ltd
The Transputer Databook 2nd edition (1989) Inmos Ltd, Inmos Document No 72 TRN 203 01
- [Inmos89c] Inmos Ltd
The Transputer Development and iq systems Databook First edition (1989) Inmos Document No 72 TRN 219 00

- [Inmos89d] Inmos Ltd
Occam 2 Toolset User Manual Inmos Document No 72
TDS 184 00 (1989)
- [Inmos90] Inmos Ltd
H1 Transputer: Product preview September (1990)
Inmos Document No 42 1473 00
- [Jackson83] M A Jackson
System Development Prentice-Hall (1983)
- [Jardine77] D A Jardine (ed)
The ANSI/SPARC DBMS Model Proc. of the 2nd
SHARE Working Conference on Data Base
Management Systems, Montreal, Canada, 26-30 April
1976, North-Holland (1977)
- [Johnson83] J R Johnson
'A prototyping success story' Datamation November
(1983) 251-256
- [Kerridge87] J M Kerridge
*'DRAT - A Proposal for a Dynamically Reconfigurable
Array of Transputers to support database applications'*
in Proc. of OUG-7 [Muntean87]
- [Kerridge89] J M Kerridge, S Wright, and R Oates
'Design, Abstract Data Types and occam' in Proc. of
OUG-10 [Bakkers89] 29-45
- [Kilo86] G Kilo
'Screen Painter' BSc Project Report, Dept of Computer
Studies, Sheffield City Polytechnic (1986)
- [Knowles89] A Knowles and T Kantchev
'Message passing in a transputer system'
Microprocessors and Microsystems Vol 13 No 2 March
(1989) 113-123
- [Knowles90] D Knowles
'Mapping a Mascot 3 design into Occam' Software
Engineering Journal Vol 5 No 4 July (1990) 207-213
- [Korson90] T Korson and J D McGregor
'Understanding Object-Oriented: A Unifying Paradigm'
Comms. of the ACM Vol 33 No 9 September (1990) 40-
60
- [Laenens88] E Laenens and D Vermeir

'An Overview of OOPS+, An Object-Oriented Database Programming Language' in [Gjessing88] Proc. of ECOOP88 350-373

- [Li90] Q Li, N Rishe, D Tal
'RISC processors in a massively parallel database machine' Microprocessors and Microsystems Vol 14 No 6 July/August (1990) 351-356
- [Lindsjorn88] Y Lindsjorn and D Sjoberg
'Database Concepts Discussed in an Object Oriented Perspective' in [Gjessing88] Proc. of ECOOP88 300-318
- [Luker86] P A Luker and A Burns
'Program generators and generation software' The Computer Journal Vol 29 No 4 (1986) 315-321
- [Mark85] L Mark and N Roussopoulos
'The New Database Architecture Framework - A Progress Report' in Information Systems: Theoretical and Formal Concepts Proc. IFIP WG8.1 Working Conference on Theoretical and Formal Aspects of Information Systems, Sitges, Barcelona, Spain, 16-18 April 1985, North-Holland (1985) 3-18
- [Mark86] L Mark and N Roussopolous
'Metadata Management' IEEE Computer Vol 19 No 12 December (1986) 26-36
- [May90] D May and P Thompson
'Transputers and Routers: Components for Concurrent Machines' in Proc. of the Occam Users Group 13th Technical Meeting 18-20 September (1990) York, UK
- [McCraken82] D D McCracken and M A Jackson
'Life Cycle Concept Considered Harmful' A C M SIGSOFT Software Engineering Notes Vol 7 No 2 April (1982) 29-32
- [Micro85] MicroFocus Ltd
Professional COBOL Language Reference Manual
MicroFocus Ltd (1985)
- [Missikof83] M Missikof and M Terranova
'The architecture of a relational database computer known as DBMAC' in [Hsiao83] 87-108
- [Muntean87] T Muntean (ed)
Proc. 7th occam User Group Meeting Grenoble, France, 14-16 September (1987)

- [Neishlos89] H Neishlos (ed)
Parallel Processing: Technology and Applications Proc.
of the International Symposium 26-28 October 1988,
Johannesburg, SA, IOS Press (1989)
- [Nelson85] K Nelson
*'Technical requirements of fourth-generation
languages'* Data Processing Vol 27 No 9 November
(1985) 12-15
- [Neuhold86] E J Neuhold
*'Objects and Abstract Data Types in Information
Systems'* in Database Semantics (DS-1) by T B Steel and
R Meersman (eds) North-Holland (1986) 1-12
- [Ng89] S Ng
'Some design issues of disk arrays' in [Compcn89] 137-
142
- [Olle78] T W Olle
The Codsyl Approach to Data Base Management John
Wiley & Sons (1978)
- [Oakley89] H Oakley
*'Mercury: an operating system for medium-grained
parallelism'* Microprocessors and Microsystems Vol 13
No 2 March (1989) 97-102
- [Patterson88] D A Patterson, G Gibson, R H Katz
*'A case for Redundant Arrays of Inexpensive Disks
(RAID)'* in [Sigmod88] 109-116
- [Patterson89] D A Patterson, P Chen, G Gibson, R A Katz
*'Introduction to Redundant Arrays of Inexpensive
Disks (RAID)'* in [Compcn89] 112-117
- [Peel89] R M A Peel
*'Issues Raised while Implementing a Layered Protocol
using Occam and the Transputer'* in Proc. of OUG-10
[Bakkers89] 152-164
- [Perihelion89] Perihelion Software Ltd
The Helios Operating System Prentice-Hall (1989)
- [Plessman88] K W Plessman and L Tassakos
*'Concurrent, object-oriented program design in real-
time systems'* Microprocessing and Microprogramming
Vol 24 (1988) 257-266

- [Poole87] F Poole
'DB4GL - An Intelligent Database System' in Research Papers of the Conference on Automating Systems Development Leicester Polytechnic, UK, April (1987)
- [Priti86] C Priti
'A Normalization Engine with a Report and Entity Restructuring Facility' BSc Project Report, Dept of Computer Studies, Sheffield City Polytechnic (1986)
- [Qiang88] X M Qiang and S Turner
'Randomized Routing : "Hot Potato" Simulations' in Proc. of OUG-9 [Askew88] 81-90
- [Ratcliff87] B Ratcliff
Software Engineering: principles and methods Blackwell Scientific Publications (1987)
- [Rishe89] N Rishe, D Tal, and Q Li
'Architecture for a Massively Parallel Database Machine' Microprocessing and Microprogramming Vol 25 (1989) 33-38
- [Roebbers89] H Roebbers and M Vlot
'A Communication Processor on the Transputer' in Proc. of OUG-10 [Bakkers89] 143-151
- [Roscoe87] A W Roscoe
'Routing messages through networks: an exercise in deadlock avoidance' in Proc. of OUG-7 [Muntean87]
- [Roussopoulos85] N Roussopoulos and L Mark
'Schema Manipulation in Self-Documenting Data Models' International Journal of Computer and Information Sciences Vol 14 No 1 (1985) 1-28
- [Salem86] K Salem and H Garcia-Molina
'Disk Striping' in Proc. of the 2nd IEEE International Conference on Data Engineering Los Angeles, USA, February (1986) 336-342
- [Schneider89] D A Schneider, D J DeWitt, S Ghandeharizadeh
'An overview of the Gamma database machine' in [Compcn89] 162-166
- [Schweppes83] H Schweppes et al
'RDBM - a dedicated multiprocessor system for database management' in [Hsiao83] 36-86
- [Senko73] M E Senko, E B Altman, M M Astrahan and P L Fehder

'Data Structures and Accessing in Data-base Systems'
IBM Systems Journal Vol 12 No 1 (1973) 64-93

- [Senko76] M E Senko
'DIAM as a Detailed Example of the ANSI SPARC Architecture' in G M Nijssen (ed) Modelling in Data Base Management Systems North-Holland (1976) 73-94
- [Sernandas87] A Sernandas, C Sernandas, and H Ehrich
'Object-Oriented Specification of Databases: An Algebraic Approach' Proc. of 13th VLDB Conference Brighton, UK, (1987) 107-116
- [Sigmod88] Proc. of the ACM SIGMOD International Conference on the Management of Data Chicago, Illinois, USA, 1-3 June 1988, ACM Press (1988)
- [Stefik84] M Stefik and D G Bobrow
'Object-Oriented Programming: Themes and Variations' The AI Magazine (1984) 40-62
- [Stonebraker86] M Stonebraker
'The Case for Shared Nothing' IEEE Database Engineering Vol 9 No 1 (1986) 4-9
- [Storer88] R Storer
'Data-driven software design using inversion' Information and Software Technology Vol 30 No 2 March (1988) 99-107
- [Stringer89] R Stringer and L C Waring
'Transputer based database organization - an example protein database implemented using pipeline and hypercube configurations' in Proc. of OUG-10 [Bakkers89] 296-300
- [Stroustrup86] B Stroustrup
The C++ Programming Language Addison-Wesley (1986)
- [Turner90] S J Turner (ed)
Tools and Techniques for Transputer Applications Proc. of OUG-12 2-4 April 1990, Exeter, UK, IOS Press (1990)
- [Tsichritzis78] D Tsichritzis and A Klug (eds)
'The ANSI/X3/SPARC DBMS Framework' Information Systems Vol 3 (1978) 173-191
- [Wasserman90] A I Wasserman, P A Pircher, and R J Muller

'The Object-Oriented Structured Design Notation for Software Design Representation' IEEE Computer Vol 23 No 3 March (1990) 50-63

- [Welch89] P H Welch
'TRANSNET - A Transputer-Based Communications Service' in Proc. of OUG-10 [Bakkers89] 198-212
- [Wiederhold86] G Wiederhold
'Views, Objects and Databases' IEEE Computer Vol 19 No 12 December (1986) 37-44
- [Williams89] R Williams and J Adkinson
'Increasing Diskette Capacity with Floptical Technology' in [Compton89] 148-150
- [Wirfs-Brock90] R J Wirfs-Brock and R E Johnson
'Surveying current research in object-oriented design' Comms. of the ACM Vol33 No 9 September (1990) 104-124
- [Xerox81] Xerox Learning Research Group
'The Smalltalk 80 System' Byte August (1981)
- [Zhao88] L Zhao and S A Roberts
'An Object-Oriented Data Model for Database Modelling, Implementation and Access' The Computer Journal Vol 31 No 2 (1988) 116-124

Appendix A

Algorithm Syntax Definition

The Occam-style syntax of the language used to describe the algorithms of P-DB4GL processes, such as, entity and schema handlers, routers, and multiplexors, is based on the definition of Occam 2 given in the Reference Manual [Inmos88b]. The Occam-style syntax is extended by the incorporation of fold lines used in the editor of the Transputer Development System (TDS) [Inmos88c]. A fold line is marked by three dots (...), and denotes a block of Occam text currently hidden from view. Fold lines, should not be confused with comment lines, marked by two hyphens (--), which are used to indicate comments ignored by a compiler. The Occam-style syntax conforms to the rules defined for comment use given in the Reference Manual. The only permitted departure, from the syntax defined by the Reference Manual and the TDS editor, is the special use of three dots (...) without a fold line name in procedure parameter lists, to denote unspecified formal channel parameters.

The following examples of procedure definitions using the Occam-style syntax demonstrate the use of fold lines, comment lines, and the special unspecified channel denotation.

```
PROC Process.with.chans (CHAN OF BYTE in, out)
  -- a comment at the start of process
  SEQ
    ... initialise fold
    -- main body now follows
    ... main body fold
  :
```

```
PROC Process.not.specified (...)
  -- an example of special use of
  -- three dots to denote unspecified formal
  -- channel parameters
  ... some constant declarations
  SEQ
    ... main body
  :
```

Appendix B

Message Formats and Occam Channels

In the development of the prototype P-DB4GL system, and also in the designs for a fully functional multiprocessor version of P-DB4GL, several different inter-process message types are used. The different message types vary considerably in respect of their typical sizes and message format. All of the messages are communicated on Occam channels, for which Occam2 channel protocols have been defined (Appendix L). Most P-DB4GL message communications conform to a higher level request-reply protocol, and require two unidirectional Occam channels to effect this communication. Some request-reply communications have different message formats, with different Occam channel protocols, for the separate request and reply messages: for example, the Filer to entity handler messages communicated on the request-reply channel pair defined by the FILER.REQ and RILER.REPLY Occam protocols. Whereas, the Basic Communication Unit (BCU) request-reply communications use the same message format (and Occam channel protocol) for messages carried on both the request and reply components of the BCU channel pair.

In the following section, the principal message types in the P-DB4GL systems are described; namely, the BCU, BCUX, BCUXR, and Filer message types. The formats of these message types are illustrated in Figure B1. The Occam 2 channel protocols used for communication of these message types are given in Appendix L. The protocol definitions use a number of system constant values, which are provided in Appendix K.

Principal P-DB4GL Message Types

BCU

The Basic Communication Unit (BCU) message is the primary message type for communication between the constituent code processes used to implement P-DB4GL applications. Each P-DB4GL code process usually has many BCU channels, and BCU channel arrays, for connection to other such processes. A BCU message is composed of the following fields:

- Eh.number, a four digit integer used to indicate the process number of a code process;

- Eh.mode, a single character used indicate the type of operation action invoked by the message ('E' - entity action, 'A' - attribute action),
- Eh.operation, a four digit integer denoting either the entity-action code invoked, or the attribute number of an attribute-action message;
- E.h.stat, a single character indicating the error status of a message;
- Eh.io, indicating whether an action is for output or input;
- Eh.val, the data component of a BCU message, including the length of the data item.

Both the request and reply messages have the same message format. The fields that are not data are considered as protocol overhead. In the current representation, each BCU message has an overhead of 12 bytes.

BCUX

The BCU-extended (BCUX) message is used to communicate messages between P-DB4GL code processes with multiplexed BCU channels. It extends the BCU message format with two additional fields:

- Source.object, an eight character field indicating the object type and identifier of the process sending the message;
- Destination.object, another eight character field indicating the object type and identifier of the intended recipient of the message.

Both the request and reply messages have the same format, and the protocol overhead is increased by 16 bytes to a total of 28 bytes per message.

BCUXR

The BCUX-ring message is used for inter-processor communication in the ring topology of the multiprocessor P-DB4GL networks. It extends the BCUX message format with three fields:

- the Source.node identifier locating the originating process of the message;
- the Destination.node identifier locating the intended recipient process of the message;
- the length of the BCUXR message in bytes.

Both the request and reply messages have the same format, and the protocol overhead is increased by 3 bytes to a total of 31 bytes per message.

Filer

The Filer messages are used in P-DB4GL applications for communication between entity handler processes and Filer processes acting as Filer Harnesses (see Appendix E) for the attached entity handlers. There are two Occam channel protocols, FILER.REQ and FILER.REPLY, defined for the request and reply messages. Both request and reply messages have variant formats; that is, each message has a preceding tag field followed by optional data fields containing key, record, or file description information. For the request message, the tag indicates the invoked Filer operation. For the reply message, the tag indicates the success or failure of the invoked operation.

In the prototype P-DB4GL test applications, Filer processes are directly connected to entity handlers without any intermediate channel multiplexing or message routing. There is no protocol overhead, only the tag field. However, in the designs for a fully functional P-DB4GL presented in Chapter 6, some Filer channels are multiplexed. This would require extensions to the existing message formats and Occam channel protocols, such as the addition of channel or process identifier fields.

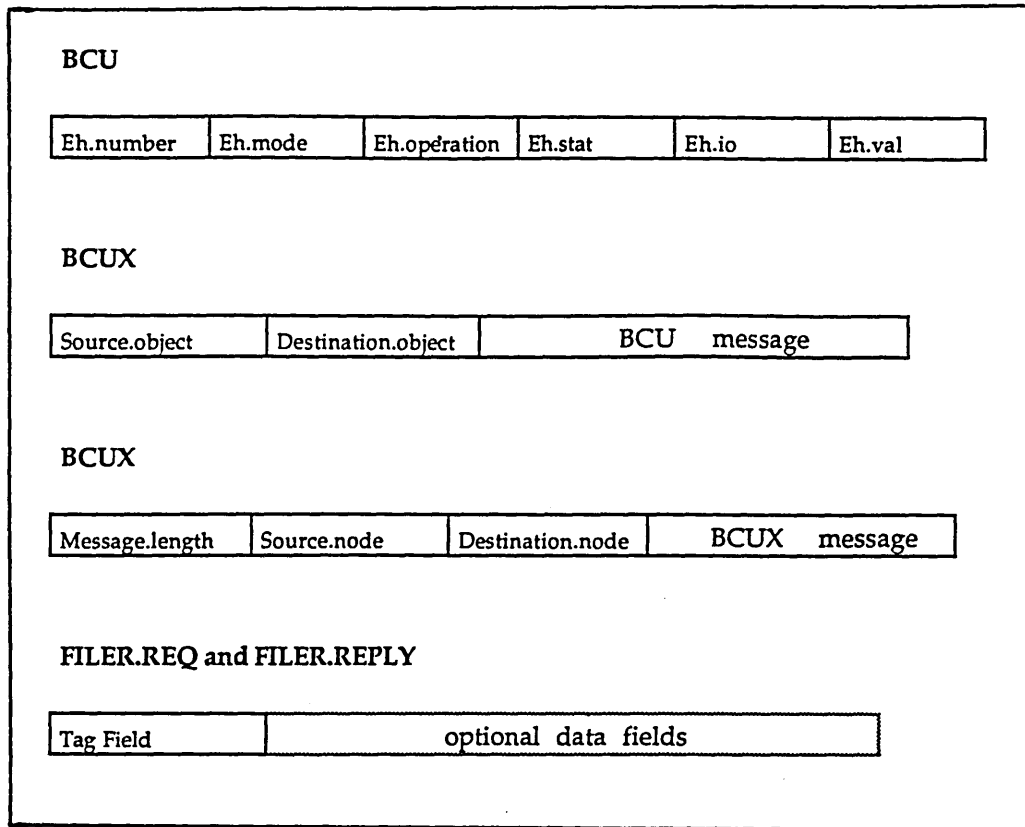


Figure B1 - P-DB4GL Message Formats

Entity and Schema Handler Algorithms

Entity Handlers and schema handlers are the data access processes used in the implementation of the generated DB4GL database applications.

Each entity handler maintains a single indexed-sequential file of records which is the physical realisation of the relation representing an Information Unit's (IU) extension. Each IU occurrence is realised as a single record in this file. All operations on IU occurrences, for example insertion, deletion, modification, are performed by the IU's entity handler. These operations are invoked by other data access processes sending a BCU request message to the entity handler. If the message is valid, the entity handler performs the requested operation and returns a reply message containing status information indicating the success or failure of the invoked operation and IU data (if requested). Full descriptions of all the operations performed by entity handlers can be found in [Ewin85a] and [Hird89]. The operations are classified into two main types:

- **entity** actions, such as STORE/READ/DELETE records (IU occurrences);
- **attribute** actions, MOVE-IN and MOVE-OUT, whereby attribute values are written to and read from the entity handler's current record (part of the entity handler's state vector, and used to hold the attribute values of the last IU occurrence read from or written to file).

A prime-view IU is supported by a **prime** entity handler and a non-prime view IU by a **coupling** entity handler. Prime and coupling entity handlers differ only slightly, both perform the same operations on IU occurrences. The only exceptions are:

- coupling entity handlers do not themselves update other entity handlers;
- a coupling entity handler will only perform an update operation when this is invoked by a message originating from its prime entity handler.

Unless directed otherwise, all references to entity handlers in the following algorithms include both prime and coupling entity handlers.

A schema handler is used to implement an application task's data access schema. A schema handler controls access to an application task's entity

handlers. Within an application task, all data access messages intended for entity handlers are first directed to, and processed by, the application task's schema handler, before being forwarded by the schema handler to the intended entity handler. The only exception to this rule is the update of coupling entity handlers by a prime entity handler. The BCU request/reply messages from the prime entity handler, invoking the update operations at the coupling entity handlers, are sent there directly, without being forwarded by the schema handler. The main processing activity performed by the schema handler is schema link realisations, that is, prefetching records (IU occurrences) related by schema links. Schema links represent a unidirectional mapping from a source attribute of one IU to a target attribute of another IU. This target attribute forms part of the primary key of the target IU. When a source attribute value is altered, which may be caused by the update of an IU occurrence, the schema link is triggered. The new attribute value is moved into the entity handler of the target attribute IU and a READ operation is invoked at the target entity handler to fetch an IU occurrence (record) with a new key value from the file.

In the sequential implementation of DB4GL, the message passing between data access processes is achieved by passing parameters between COBOL code modules at code module call and returns. For P-DB4GL, in which data access processes are implemented as concurrent Occam processes, message passing is performed using inter-process Occam channels and the Occam communication primitives to send and receive messages on these channels. In the algorithms presented below, the particular mechanisms used to effect this message passing, for example formal channel parameters on Occam procedure definitions, have been omitted for reasons of clarity.

Four different versions of the P-DB4GL data access processes (prime entity handler, coupling entity handler, and schema handler) have been implemented. Each of the four versions is functionally equivalent to, and has the same interface as, the other three versions; but the algorithm encapsulated inside each handler process is different. That is, all versions are identical in what they do, but each version differs in how they do it.

These data access processes have been used to construct many different test applications. In all of the test applications, each constituent data access process executes concurrently and independently, interacting with other data access processes solely by message passing. However, the presence of this

inter-process concurrency does not in itself produce performance improvements when the test applications are executed on parallel transputer hardware. Alterations to the purely sequential algorithms contained within the data access processes can affect the inter-process concurrent behaviour and hence improve overall performance; but it is the incorporation of intra-process concurrency to the data access processes that is crucial to obtaining maximum performance improvement. The reason for the different versions of the data access handlers is to examine the precise effect of this intra-process concurrency on test application performance, whilst maintaining the same functionality.

The algorithms for the "original" (version 1) P-DB4GL handlers are based on the descriptions of the DB4GL entity and schema handlers given in [Ewin85a]. These "original" sequential algorithms are reproduced here in pseudocode, along with their equivalent version 1 P-DB4GL data access processes described using the Occam-style syntax defined in Appendix A. The "modified" sequential (version 2) and the concurrent version 3 and version 4 data access processes are also described using this Occam-style syntax.

The "original" (version 1) entity and schema handler algorithms are:

```

PROCEDURE Entity.handler
  initialise
  WHILE running
    receive request message to entity
    validate message
    IF valid message for this entity THEN
      perform the invoked operation
      IF couple updates triggered THEN
        perform coupling entity updates
      ENDIF
    ENDIF
    send reply message from entity
    check for entity handler termination
  ENDWHILE
ENDPROC

```

```

PROCEDURE Schema.handler
  initialise
  WHILE running
    receive request message to schema
    validate message
    IF valid message for a schema entity THEN
      send request message to entity
      receive reply message from entity
      process reply message from entity
    ENDIF
    send reply message from schema
    check for schema handler termination
  ENDWHILE
ENDPROC

```

The equivalent P-DB4GL version 1 data access processes, described using the Occam-style syntax are:

```

PROC Entity.handler.v1 (...)
  -- "original" sequential algorithm
  SEQ
    ... initialise
  WHILE running
    SEQ
      ... receive request message to entity
      ... validate message
    IF
      valid.message
      SEQ
        ... perform the invoked operation
      IF
        couple.updates.triggered
          ... perform coupling entity updates
        TRUE -- else
          SKIP
      TRUE -- else
        SKIP
    ... send reply message from entity
    ... check for entity handler termination
  :

```

```

PROC Schema.handler.v1 (...)
  -- "original" sequential algorithm
  SEQ
    ... initialise
  WHILE running
    SEQ
      ... receive request message to schema
      ... validate message
    IF
      valid.message
        SEQ
          ... send request message to entity
          ... receive reply message from entity
          ... process reply message from entity
        TRUE -- else
          SKIP
      ... send reply message from schema
      ... check for schema handler termination
  :

```

In the sequential implementation of DB4GL, in which an application task is constructed from a number of separately compiled COBOL code modules linked together in a hierarchical control structure, only one code module can be executing at any given time. The exact ordering of instructions within a handler's algorithm is not always important, different sequences of instructions can often perform the same function and the overall execution time of the application task remains the same. In P-DB4GL, the ordering of instructions within a handler's algorithm can have a marked effect on the overall execution time of an application task, inefficient sequential algorithms can leave concurrent processes idle and unable to do any useful processing.

The constituent data access processes in a P-DB4GL application task execute concurrently, but certain sequences of instructions introduce processing delays that inhibit inter-process concurrent processing thus impairing overall application performance. In the "original" sequential version of the schema handler, the schema reply message is not returned from the schema handler until the entity handler reply message has been processed and any necessary schema link realisation have been performed. However, as the schema reply message is unaffected by the schema link processing, an unnecessary delay is introduced to the P-DB4GL version 1 data access process when the sequence of instructions in the "original" algorithm is maintained.

In the "modified" sequential (version 2) schema handler process, the schema reply message is sent prior to the schema link processing. This increases the amount of effective inter-process concurrent processing and improves overall performance because the schema handler and user processes "driving" the schema handler can both be processing concurrently. The schema link processing can be performed by the version 2 handler in the time interval before the arrival of the next schema request message. The version 1 algorithm caused the schema handler to remain idle during this time interval. A similar alteration is made to the P-DB4GL version 2 entity handlers, allowing the entity reply message to be sent to the schema handler before any coupling entity updates are performed. Thus enabling schema, prime entity, and coupling entity handlers to all perform useful processing concurrently.

```

PROC Entity.handler.v2 (...)
  -- "modified" sequential algorithm
  SEQ
    ... initialise
  WHILE running
    SEQ
      ... receive request message to entity
      ... validate message
      IF
        valid.message
          SEQ
            ... perform the invoked operation
            ... send reply message from entity
          IF
            couple.updates.triggered
              ... perform couple entity updates
            TRUE -- else
              SKIP
          TRUE -- else, invalid message
            ... send reply message from entity
      ... check for entity handler termination

```

```

PROC Schema.handler.v2 (...)
-- "modified" sequential algorithm
SEQ
... initialise
WHILE running
  SEQ
... receive request message to schema
... validate message
  IF
    valid.message
      SEQ
... send request message to entity
... receive reply message from entity
... send reply message from schema
... process reply message from entity
  TRUE -- else, invalid message
... send reply message from schema
... check for schema handler termination
:

```

In the version 3 data access processes intra-process concurrency is introduced. The entity handler process is modified to permit state vector inspection to occur concurrently with coupling entity update. The state vector inspection procedure allows the entity handler to continue to receive and process BCU request/reply messages from the schema handler, so long as the messages only request to read the entity handler's state variables. The state variables contain the attribute values of the current IU occurrence, that is, the last record read from or written to the entity handler's file. When a message is received invoking an operation which would update the entity handler's state, this message is blocked, and the state vector procedure terminates. When the coupling entities have been updated, the blocked message is processed as usual. A similar state vector inspection procedure is added to the version 3 schema handler process to permit read only access to a restricted number of entity handler's states during schema link realisation.

```
PROC Entity.handler.v3 (...)  
-- concurrent algorithm  
SEQ  
... initialise  
WHILE running  
  SEQ  
    IF  
      NOT message.blocked  
        ... receive request message to entity  
        message.blocked  
        SKIP -- message already received by sv inspection  
    ... validate message  
    IF  
      valid.message  
        SEQ  
          ... perform the invoked operation  
          ... send reply message from entity  
          IF  
            couple.updates.triggered  
              PAR  
                ... perform couple entity updates  
                ... entity state vector inspection  
            TRUE -- else, updates not triggered  
              SKIP  
          TRUE -- else, invalid message  
            ... send reply message from entity  
        ... check for entity handler termination  
:
```

```

PROC Schema.handler.v3 (...)
  -- concurrent algorithm
  SEQ
    ... initialise
  WHILE running
    SEQ
      IF
        NOT message.blocked
          ... receive request message to schema
        message.blocked
          SKIP -- message already received by sv inspection
      ... validate message
    IF
      valid.message
        SEQ
          ... send request message to entity
          ... receive reply message from entity
          ... send reply message from schema
          ... check for schema link activation
        IF
          schema.links.triggered
            PAR
              ... schema link realisation
              ... state vector inspection
            TRUE --else, links not triggered
            SKIP
          TRUE -- else, invalid message
            ... send reply message from schema
            ... check for schema handler termination
      :

```

In the version 4 data access processes there are separate procedures for prime and coupling entity handlers. The prime entity handler is modified to incorporate concurrent update of its coupling entity handlers - this is indicated by the replicated PAR construct. The coupling entity handler allows state vector inspection to occur concurrently with update by its prime entity handler. The schema handler has a modified schema link realisation procedure which allows some of the links to be realised concurrently.

```

PROC Prime.Entity.handler.v4 (...)
-- concurrent algorithm
SEQ
... initialise
WHILE running
  SEQ
  IF
    NOT message.blocked
    ... receive request message to entity
    message.blocked
    SKIP -- message already received by sv inspection
  ... validate message
  IF
    valid.message
    SEQ
    ... perform the invoked operation
    ... send reply message from entity
    IF
      couple.updates.triggered
      PAR
        PAR i = 0 FOR no.of.couples
          ... update coupling entity i
          ... entity state vector inspection
        TRUE -- else, updates not triggered
        SKIP
      TRUE -- else, invalid message
    ... send reply message from entity
  ... check for entity handler termination
  :

```

```

PROC Coupling.Entity.handler.v4 (...)
-- concurrent algorithm
SEQ
... initialise
WHILE running
  SEQ
  IF
    NOT message.blocked
    ... receive request message to entity
    message.blocked
    SKIP -- message already received by sv inspection
  ... validate message
  IF
    valid.message.from.schema
    SEQ
    ... perform the invoked operation
    ... send reply message from entity
    valid.message.from.prime
    PAR
    ... process prime messages until update completed
    ... state vector inspection by schema
  TRUE -- else, invalid message
  ... send reply message from entity
  ... check for entity handler termination
  :

```



```
PROC Schema.handler.v4 (...)  
-- concurrent algorithm  
SEQ  
... initialise  
WHILE running  
  SEQ  
  IF  
  NOT message.blocked  
    ... receive request message to schema  
  message.blocked  
    SKIP -- message already received by sv inspection  
  ... validate message  
  IF  
  valid.message  
    SEQ  
    ... send request message to entity  
    ... receive reply message from entity  
    ... send reply message from schema  
    ... check for schema link activation  
    IF  
    schema.links.triggered  
      PAR  
      PAR i = 0 FOR no.of.links  
        ... realise link i when ready  
        ... state vector inspection  
      TRUE --else, links not triggered  
      SKIP  
    TRUE -- else, invalid message  
    ... send reply message from schema  
  ... check for schema handler termination  
:
```

Appendix D

Disc and Filer Algorithms

A Filer process is used to support the filing requirements of the entity handlers in the P-DB4GL test applications. Each entity handler is connected by a pair of Occam channels to a single Filer process which acts as a Filer Harness for the attached entity handler. The behaviour of a Filer Harness (FH) is defined in by the Filer Harness Specification (Appendix E). Each Filer process is connected by another channel pair to a Disc process. A single Disc process may be connected to many Filer processes. The set of all Filer and Disc processes used within a particular P-DB4GL test application collectively represent that test application's filing simulation. The algorithms for the Filer and Disc processes are described below using the Occam-style syntax defined in Appendix A.

There are two versions of the Filer process used in the P-DB4GL test applications: Filer.v1 and Filer.v2. Both versions function as an FH in an identical manner and provide their attached entity handler with an Indexed-Sequential (IS) file as defined by the Filer Harness Specification. However the Filer process versions differ in their simulated disc access behaviour. For Filer.v1 it is assumed that all filing operations, invoked by the attached entity handler on the FILER.REQ channel, have an equal probability of disc access. Consequently, each invoked filing operation always results in a communication on the Disc channel pair and causes an associated simulated disc access delay.

```
PROC Filer.v1 (CHAN OF FILER.REQ request.in,
              CHAN OF FILER.REPLY reply.out,
              CHAN OF BYTE disc.out, disc.in)
  -- "stores" records in an array in memory
  SEQ
    ... initialise
  WHILE running
    SEQ
      ... receive filing request message
      ... perform invoked filing operation
      -- simulate disc access
    SEQ
      ... output to Disc process
      ... input from Disc process
      ... send filing reply message
      ... check for process termination
:
```

Filer.v2 operates with slightly different assumptions about the likelihood of disc access when filing operations are invoked. For Filer.v2, each filing operation has a different probability of disc access. The Open and Close operations both have a 100% probability of disc access. The Read operation has a 90% probability. The Write, Rewrite, and Delete operations all have a 50% probability. The Read-Next operation has a 10% probability. All Start operations have a 90% probability. The Terminate operation, because it is not strictly a filing operation, communicates a special terminate message with no associated disc access delay to the Disc process.

```

PROC Filer.v2(CHAN OF FILER.REQ request.in,
              CHAN OF FILER.REPLY reply.out,
              CHAN OF BYTE disc.out,disc.in)
-- "stores" records in array an in memory
-- with various probabilities of "disc access"
SEQ
  ... initialise
  WHILE running
    SEQ
      ... receive filing request message
      ... perform invoked filing operation
      -- simulate disc access
    SEQ
      ... calculate disc access probability
    IF
      disc.access.indicated
        SEQ -- access disc
          ... output to Disc process
          ... input from Disc process
      TRUE -- else
        SKIP -- no disc access required
      ... send filing reply message
      ... check for process termination
  :

```

The reason for assigning probabilities to the likelihood of disc access for each filing operation is to more accurately simulate the behaviour of genuine filing processes. It is assumed that in a fully functional P-DB4GL application, operating with realistic database sizes, it is unlikely that an entire database file could be held in the memory of a single processor. The number of data and index records processed by a genuine filing process (comprising the information unit extension belonging to a single entity handler) would be so large as to be mostly stored on disc, with only a small fraction of the total temporarily held in buffers in a processor's memory. As a consequence of this buffering it is likely that for some invoked filing operations disc access will not be necessary, because the required index and data records will already be held in a buffer in memory. It is further assumed that because of

the index-sequential nature of the files, some file operations (such as a Read-Next) have a greater probability of finding the required data in a buffer, whereas other operations (such as a random Read) have a lower probability and will more frequently need disc access.

It must be noted that the simulated disc access behaviour is not based the measured performance of genuine filing software. Clearly the performance of such software depends on so many factors such as: type and efficiency of algorithms; number and size of buffers; memory size and interface; type of disc and interface. Consequently, no inferences concerning the absolute performance figures for fully functional P-DB4GL applications can be drawn from the performance figures obtained from the prototype P-DB4GL test applications using the simulated filestore. However, the simulated filestore can be used to make valid conclusions about the relative performance of different versions of the data access processes used in P-DB4GL test applications.

```

PROC Disc([]CHAN OF BYTE disc.in, disc.out,
          VAL INT no.of.filers, disc.delay)
  SEQ
  ... initialise
  WHILE running
    -- wait for "disc access" request from a Filer
    ALT i = 0 FOR no.of.filers
      disc.in[i] ? disc.request -- accept disc access
      SEQ -- perform "disc access"
        IF
          (disc.request = disc.terminate)
            ... check for Disc process termination
          TRUE -- else
            -- perform timed delay
            Delay(disc.delay)
        disc.out[i] ! disc.reply -- release disc
  :
```

The Disc process represents a very crude simulation of disc access by Filer processes in the P-DB4GL test applications. A key feature of the Disc process is the facility to alter the Filer-Disc connections within a test application in order to simulate the effect of different mappings of files to discs. Although the Disc process has no provision for storing data, it does simulate contention over the read/write mechanism of a genuine disc - only one of the Filer processes connected to a Disc process can be accessing the disc at any given time. A simple fixed size timed delay simulates the mechanical properties of head movements and rotational delay. The size of this timed delay is determined by a VAL parameter "disc.delay" in the Occam

procedure for the Disc process; this is an integer value in timer ticks (one tick is 64 microseconds for processes executing at low priority). Typical values used in the test applications are 0, 250, 500, and 1000 ticks (that is, 0, 16, 32, and 64 milliseconds). The Disc process terminates when all of its connected Filer processes have sent it a non-disc access termination message.

Filer Harness Specification

The filer harness specification describes a persistent object, known as a Filer Harness (FH) which provides filing functions for P-DB4GL database applications. Specifically, one FH supports the filing requirements of a single entity handler. In the prototype P-DB4GL system, a simulation of a multiple disc filestore has been implemented using Filer and Disc processes. The Filer process serves as an FH in the P-DB4GL test applications, and with a few minor restrictions conforms to the filer harness specification. In a fully functional P-DB4GL system, the filestore simulation will be replaced by genuine discs and filing software which will provide an FH for each entity handler. The filer harness specification only defines the function and interface of the filing requirements for an entity handler, it does not specify any physical or temporal characteristics.

An FH provides one unnamed Index-Sequential (IS) file. The IS file stores fixed length records, each record identified by a single fixed length key. The FH provides: operations to read, write, delete, and amend records; operations to effect the current record pointer used in the control of sequential reads; operations to open and close the IS file; and a terminate operation to halt the execution of the FH. The behaviour of the FH IS file is modelled on the IS file COBOL files [Micro85] used in the sequential implementation of the DB4GL system [Ewin85a] [Hird89] from which the Parallel-DB4GL system is derived. The COBOL IS file is more complicated than the FH IS file which incorporates only a subset of the COBOL IS file behaviour.

The FH operations can only be invoked by message passing. The representation and implementation details of the FH are hidden from the software objects that use it. When an entity handler interacts with an FH it must conform to a strict request-reply message protocol. The entity handler sends a request message to the FH indicating which operation it requires. The FH responds to this request and returns a reply message which informs the entity handler of the success or failure of the invoked operation and contains data if requested.

The Filer process used as an FH in the P-DB4GL test applications is implemented in Occam, and two Occam channels are used to support the

request-reply message interaction between the Filer process and entity handler. These channels are defined by two Occam tagged protocols, the FILER.REQ and FILER.REPLY protocols. Messages conforming to the FILER.REQ protocol have a tag field indicating the FH operation requested by the entity handler, followed by appropriate data fields. Reply messages conforming to the FILER.REPLY protocol have a tag field indicating the error status of the invoked FH operation followed by appropriate data fields if the operation is successful. Successful operation messages have protocol tags suffixed with ".ok". Failed operation messages have one of three error tags: F.at.end, F.invalid.key, F.error. It is not essential for the Filer process, or any other software object acting as an FH, to be implemented in Occam. However, the FH must be capable of interfacing with the entity handler via Occam channels conforming to the FILER.REQ and FILER.REPLY protocols.

In order to aid system development and testing, a number of variant forms of FH are permitted (and variant Filer processes have been implemented). In particular, some FH variants are capable of initialising themselves with internally generated test data. When initialised, these test data FH's generate a file of dummy records conforming to the appropriate file description but with random field values.

When an FH is executed, it first initialises itself with test data (if required); then, sets the current record pointer to undefined; and, in a closed state, waits to receive a message from the filer request channel invoking an FH operation. While it is in its closed state, only the open and terminate operations can be successfully invoked in the FH, all other operations will fail and return an error message. The FH will continue to receive operation requests and send replies until a terminate message halts the FH execution.

The operations supported by the FH are listed below. Each operation has its name and parameter list followed by the FILER.REQ protocol message that invokes it, and a description of the operation.

open <i.o.status> <record.length> <key.length>

F.open; BYTE; INT; INT

If the FH is currently open, the open operation fails and an error message (F.error) is returned.

If the FH is currently closed, the record description and io status are checked, an invalid record description or an invalid io status will result in an error (F.error) being returned. If the record description and io status are valid, the current record pointer is positioned to point to the first existing record in key sequence, if no records exist, the current record pointer points to the end of file. The FH is successfully opened and an open ok (F.open.ok) message is returned.

A record description is invalid if:

(record.length <= 0) OR
 (key.length <= 0) OR
 (key.length > record.length) OR
 (record.length > max.record.length) OR
 (key.length > max.key.length)

otherwise it is valid. The maximum key length is 120, this is a limit imposed in the COBOL implementation of DB4GL.

The io status is valid only if it has one of two permitted values: 'I' or 'O'. 'I' is an abbreviation for input.mode, this opens the FH in a state that permits read only access to its records. 'O' is an abbreviation for input.output.mode, this opens the FH in a state that permits read and write access to the records.

Note that, the FH open operation is similar to the COBOL OPEN a file with DYNAMIC ACCESS MODE operation, i.e. both sequential and random (Indexed) access is permitted. However, ALTERNATE RECORD keys are not permitted.

close

F.close

If the FH is currently closed, the close operation fails and an error message is returned (F.error).

If the FH is currently opened, the FH is successfully closed and a successful close message (F.close.ok) is returned.

read <key>

F.read; INT::[]BYTE

If the FH is currently closed, the read operation fails and an error message is returned (F.error).

If the FH is currently opened, the stored record with a key value that is equal to the supplied key value is located, the current record pointer is updated to point to the located record and a successful read message with the located record value (F.read.ok; INT::[]BYTE) is returned. If a stored record cannot be located with the supplied key, the read operation has failed, the current record pointer is undefined and an error message (F.invalid.key) is returned.

read next

```
F.read.next
```

If the FH is currently closed, the read next operation fails and an error message is returned (F.error).

If the FH is currently opened, check the current record pointer, if it is undefined then the read next operation has failed and an error message (F.error) is returned. If the last operation to position the current record pointer was either a read, read next or delete operation, then update the current record pointer to point to the next stored record in key sequence, if a record cannot be located the current record pointer is updated to end of file. If the current record pointer is at end of file, then the read next operation has failed, the current record pointer is undefined and an error message (F.at.end) is returned. If the current record pointer is not at end of file, the read next operation succeeds and a successful message with the current stored record value (F.read.next.ok; INT::[]BYTE) is returned.

write <record>

```
F.write; INT::[]BYTE
```

If the FH is currently closed or the FH is currently opened with io status of 'T' (read only access), the write operation fails and an error message (F.error) is returned.

If the FH is currently open, check the validity of the supplied record. If the supplied record is invalid or a stored record already exists with the same key as the supplied record's key, then the write operation fails and an error message (F.invalid.key) is returned. If the supplied record is valid and no stored record already exists with the same key as the supplied record, the write operation succeeds, the supplied record becomes one of the stored records and a successful write message (F.write.ok) is returned.

rewrite <record>

```
F.rewrite; INT::[]BYTE
```

If the FH is currently closed or the FH is currently opened with io status of 'T' (read only access), the rewrite operation fails and an error message (F.error) is returned.

If the FH is currently open, check the validity of the supplied record. If the supplied record is invalid or a stored record with the same key as the supplied record's key does not already exist, then the rewrite operation fails and an error message (F.invalid.key) is returned. If the supplied record is valid and a stored record already exists with the same key as the supplied record, the rewrite operation succeeds, the supplied record replaces the stored record of the same key and a successful rewrite message (F.rewrite.ok) is returned.

delete <key>

```
F.delete; INT::[]BYTE
```

If the FH is currently closed or the FH is currently opened with io status of 'T' (read only access), the delete operation fails and an error message (F.error) is returned.

If the FH is currently open, check the validity of the supplied key. If the supplied key is invalid or a stored record with the same key as the supplied key does not already exist, then the delete operation fails and an error message (F.invalid.key) is returned. If the supplied key is valid and a stored record already exists with the same key as the supplied key, the delete operation succeeds, the stored record is removed from the FH, the supplied key value is now available for subsequent writes to the FH, and a successful delete message (F.delete.ok) is returned.

start equal <key>

```
F.start.equal; INT::[]BYTE
```

If the FH is currently closed, the start equal operation fails and an error message (F.error) is returned.

If the FH is currently opened, check the validity of the supplied key. If the supplied key is invalid, the operation has failed, the current record pointer is undefined and an error message (F.invalid.key) is returned. If the supplied key is valid, the current record pointer is positioned to the stored record whose key is equal to the supplied key, and a successful start message (F.start.ok) is returned: if no such stored record exists, then the operation fails, the current record pointer is undefined and an error message (F.invalid.key) is returned.

start greater <key>

```
F.start.greater; INT::[]BYTE
```

If the FH is currently closed, the start greater operation fails and an error message (F.error) is returned.

If the FH is currently opened, check the validity of the supplied key. If the supplied key is invalid, the operation has failed, the current record pointer is undefined and an error message (F.invalid.key) is returned. If the supplied key is valid, the current record pointer is positioned to the first stored record whose key is greater than the supplied key, and a successful start message (F.start.ok) is returned: if no such stored record exists, then the operation fails, the current record pointer is undefined and an error message (F.invalid.key) is returned.

start not less <key>

```
F.start.not.less; INT::[]BYTE
```

If the FH is currently closed, the start not less operation fails and an error message (F.error) is returned.

If the FH is currently opened, check the validity of the supplied key. If the supplied key is invalid, the operation has failed, the current record pointer is undefined and an error message (F.invalid.key) is returned. If the supplied key is valid, the current record pointer is positioned to the first stored record whose key is not less than the supplied key, and a successful start message (F.start.ok) is returned: if no such stored record exists, then the operation fails, the current record pointer is undefined and an error message (F.invalid.key) is returned.

terminate

```
F.terminate
```

When the FH receives this message it checks to see if it is in a valid state to allow termination - currently there are no states of the FH for which termination is defined to be invalid. If the FH is in a valid state for termination, then a successful termination message (F.terminate.ok) is returned and execution of the FH is halted, otherwise an error message (F.error) is returned.

Representation of Records and Keys for the Filer Harness Channel Protocols

Records and keys are represented in the two FH protocols, FILER.REQ and FILER.REPLY, as ASCII byte strings. Key strings are of length 1 to max.key.length, record strings are of length 1 to max.record.length. Every record has a key part, that is, for any given record, a substring of the record from the zeroth byte (or character) for key.length number of characters is the key part of that record. The remainder of the record is the data part. Records may consist solely of key part (i.e. the data part is of length zero), but records must be at least as long as their key part. All the stored records in an FH must be of an equal and fixed length. This length is fixed when the FH is first opened and is determined by the record length value of the record description parameter. Only keys of equal length can be compared.

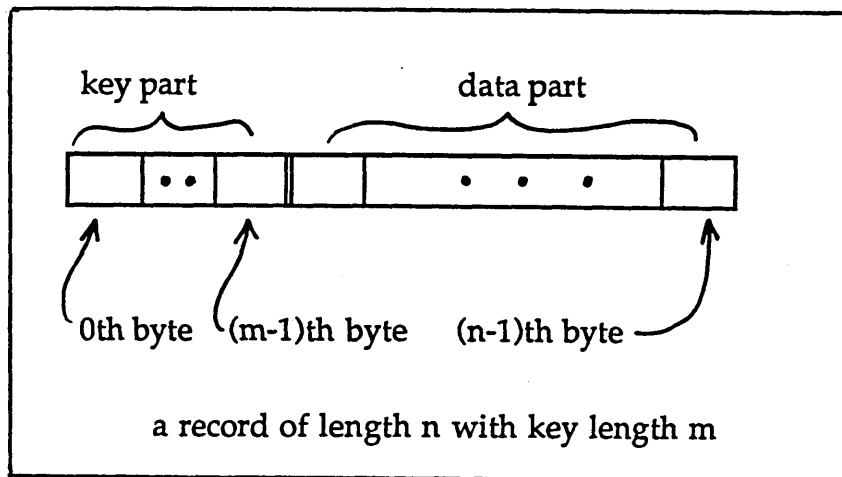


Figure E1 - Filer Harness Record Representation

Occam 2 Protocol and Constant Declarations required for Filer Harness Implementation

```

-- ***** Constants *****
VAL INT max.attribute.value.length IS 80 :
VAL INT max.no.of.fields IS 20 :
VAL INT max.no.key.fields IS 19 :
VAL INT max.record.length IS (max.no.of.fields *
                               max.attribute.value.length) :
VAL INT max.key.length IS 120 :
VAL BYTE input.mode IS 'I' :
VAL BYTE input.output.mode IS 'O' :
VAL BYTE null.char IS 0 (BYTE) :
VAL BYTE numeric IS '9' :
VAL BYTE character IS 'X' :
-- ***** Filer Protocols *****
-- <file.description> = <i.o.status> <record.description>
-- <i.o.status> = BYTE
-- <record.description> = <record.length> <key.length>
-- <record.length> = INT
-- <key.length> = INT
-- <key> = INT::[]BYTE
-- <record> = INT::[]BYTE
PROTOCOL FILER.REQ
CASE
  F.open; BYTE; INT; INT          -- <file.description>
  F.close
  F.read; INT::[]BYTE            -- <key>
  F.read.next
  F.write; INT::[]BYTE          -- <record>
  F.rewrite; INT::[]BYTE        -- <record>
  F.delete; INT::[]BYTE         -- <key>
  F.start.equal; INT::[]BYTE    -- <key>
  F.start.greater; INT::[]BYTE -- <key>
  F.start.not.less; INT::[]BYTE -- <key>
  F.terminate
:
PROTOCOL FILER.REPLY
CASE
  F.open.ok
  F.close.ok
  F.read.ok; INT::[]BYTE        -- <record>
  F.read.next.ok; INT::[]BYTE   -- <record>
  F.write.ok
  F.rewrite.ok
  F.delete.ok
  F.start.ok
  F.terminate.ok
  F.at.end
  F.invalid.key
  F.error
:

```

Appendix F

User Process Test Harness

The User Process Test Harness (User Harness) has been used to supply P-DB4GL test applications with test data. The test data is supplied in the form of Basic Communication Unit (BCU) messages sent out and received on a pair of Occam channels conforming to the BCU request-reply protocol. This BCU channel interface is identical to that of a P-DB4GL User Process and the User Harness has been used as a substitute for genuine User Processes in the P-DB4GL test applications.

The User Harness exists in a number of variant forms. There are versions for execution on different processor types and different transputer boards. There are two modes of operation for the User Harness: **interactive mode**, whereby single BCU messages are composed at a terminal then transmitted to and received from the test application; and **batch mode**, in which a set of test data already stored in the User Harness, and composed of many BCU messages, is run through the test application. All versions of the User Harness record the processing time for each BCU message (measured as the time interval between sending a BCU-request message and receiving the corresponding BCU-reply message).

The "stand-alone" versions of the User Harness are only capable of interactive operation. However, they are small programs and can execute on processors with very little external memory (ie less than 64K). They can only measure the processing time of each individual BCU message composed and sent to the test application, and have no facility to file a trace of the messages sent.

The "EXE" versions of the User Harness can operate in either interactive or batch mode, and can store a trace file of the BCU request-reply message contents and processing times. When operated in batch mode they can also record and file a trace of the transaction processing times. Each test data set for batch mode operation is composed of several "transactions", or significant database operations (such as Store a record). These "transactions" are decomposed into several smaller "atomic" actions, each represented by a BCU message (Figure F1).

A sequence of transactions can be optimised to reduce the total number of BCU messages communicated (Figure F2). The P-DB4GL User Processes typically communicate highly optimised sequences of BCU messages in which it would be difficult to discern the transaction boundaries from an examination of the BCU message stream. It must be noted that the "transaction" used in the testing of P-DB4GL applications is not an indivisible collection of updates identified for the purposes of locking and recovery, it is merely a convenient way of referring to a significant database operation.

An additional feature of the User Harness, when operated in batch mode, is the facility to introduce timed delays into the BCU message stream transmitted during a test run. There are two sorts of delay: inter-BCU, and inter-transaction. An inter-BCU delay is inserted between the receipt of a BCU-reply message and the transmission of the next BCU-request message in the test data set. Inter-transaction delays are inserted between transaction boundaries marked in the test data sequence. The purpose of these inter-BCU and inter-transaction delays is to simulate the delays associated with processing load and user response occurring in genuine P-DB4GL User Processes. The presence of these delays has a marked effect on the performance of the data access processes. Figure F3 illustrates the processing and idle times for a User Harness and versions 1 and 2 of a schema handler during a sequence of BCU messages; and Figure F4 shows the combined effect of inter-BCU and inter-transaction delays upon these times. However, the User Harness does not fully simulate genuine User Processes; because during a timed delay the User Harness is descheduled on a transputer and does not load a processor in the same way as a genuine User Process would.

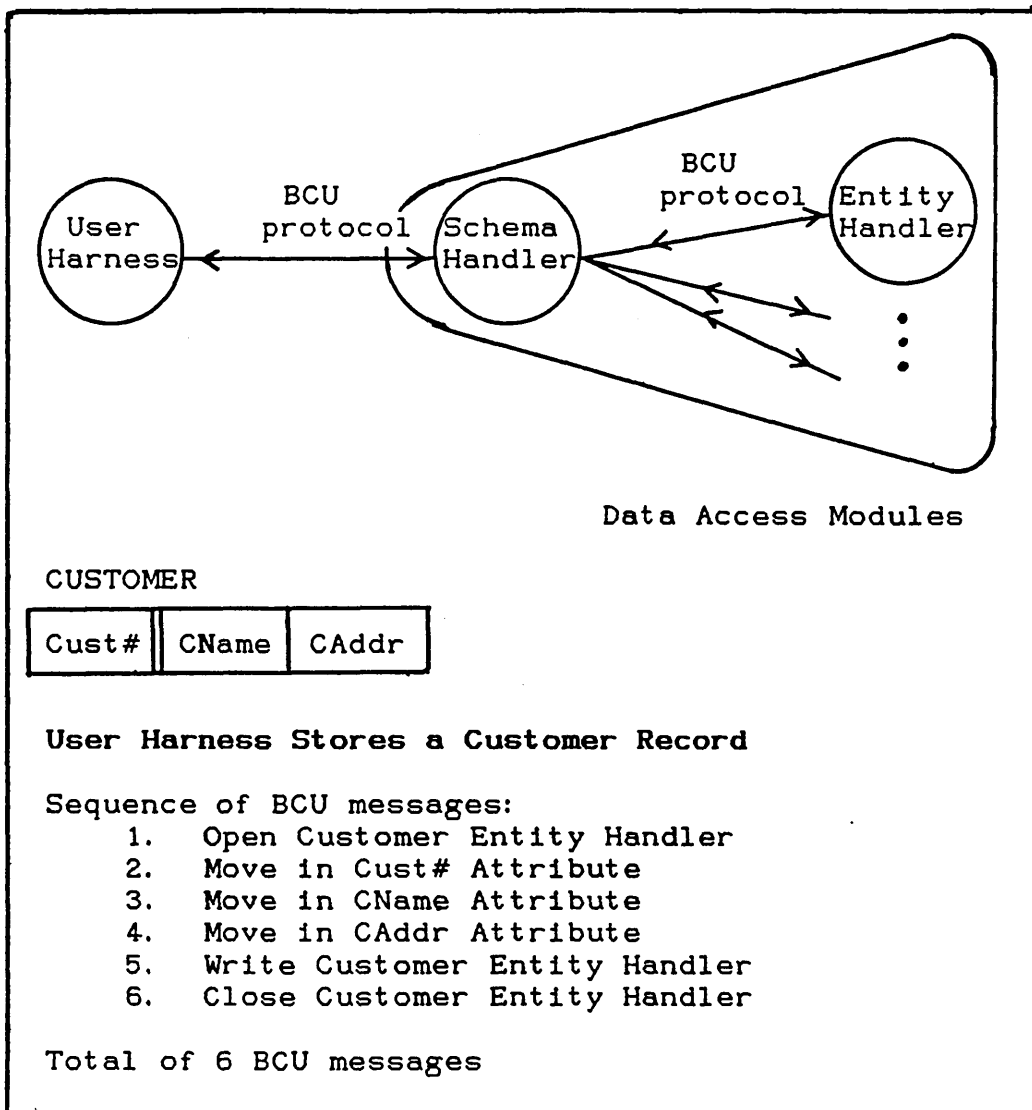


Figure F1 - A Single Transaction

**User Harness Stores, Updates, and Deletes
a Customer Record**

Non-optimised Sequence of Transactions:

Transaction	No of BCU messages
T1 Store Record	6
1. Open Customer	
2. Move in Cust#	
3. Move in CName	
4. Move in CAddr	
5. Write Customer	
6. Close Customer	
T2 Update Record	7
1. Open Customer	
2. Move in Cust#	
3. Read Customer	
4. Move out CAddr	
5. Move in (updated) CAddr	
6. Rewrite Customer	
7. Close Customer	
T3 Delete Record	4
1. Open Customer	
2. Move in Cust#	
3. Delete Customer	
4. Close Customer	

Total of 17 BCU messages

Optimised Sequence of Transactions:

Transaction	No of BCU messages
T1 Store Record	5
1. Open Customer	
2. Move in Cust#	
3. Move in CName	
4. Move in CAddr	
5. Write Customer	
T2 Update Record	4
1. Read Customer	
2. Move out CAddr	
3. Move in (updated) CAddr	
4. Rewrite Customer	
T3 Delete Record	2
1. Delete Customer	
2. Close Customer	

Total of 11 BCU messages

Figure F2 - Optimised Sequence of Transactions

A sequence of BCU messages sent from a User Harness to version 1 and 2 Schema Handlers, showing the proportions of processing and idle time

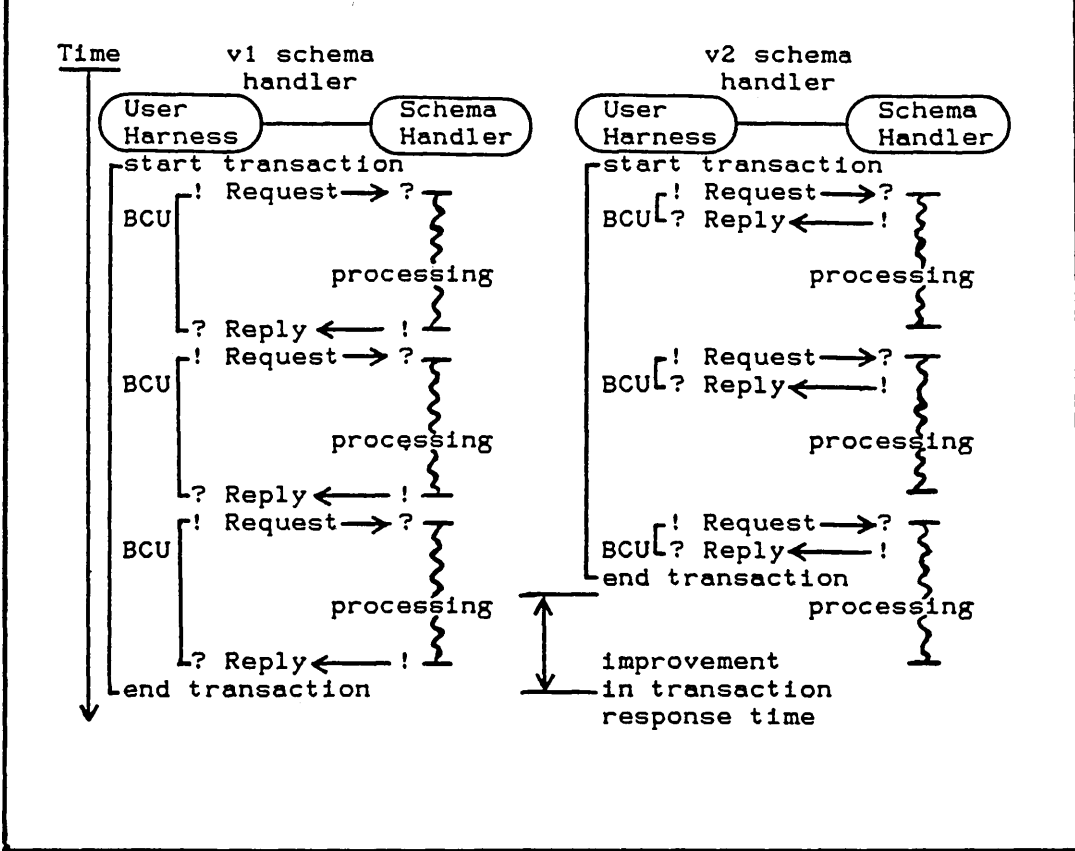


Figure F3 - A Sequence of BCU Messages

Effect of inter-BCU and inter-transaction delays in the User Harness simulation on the processing time of a sequence of transactions for the v1 and v2 schema handlers

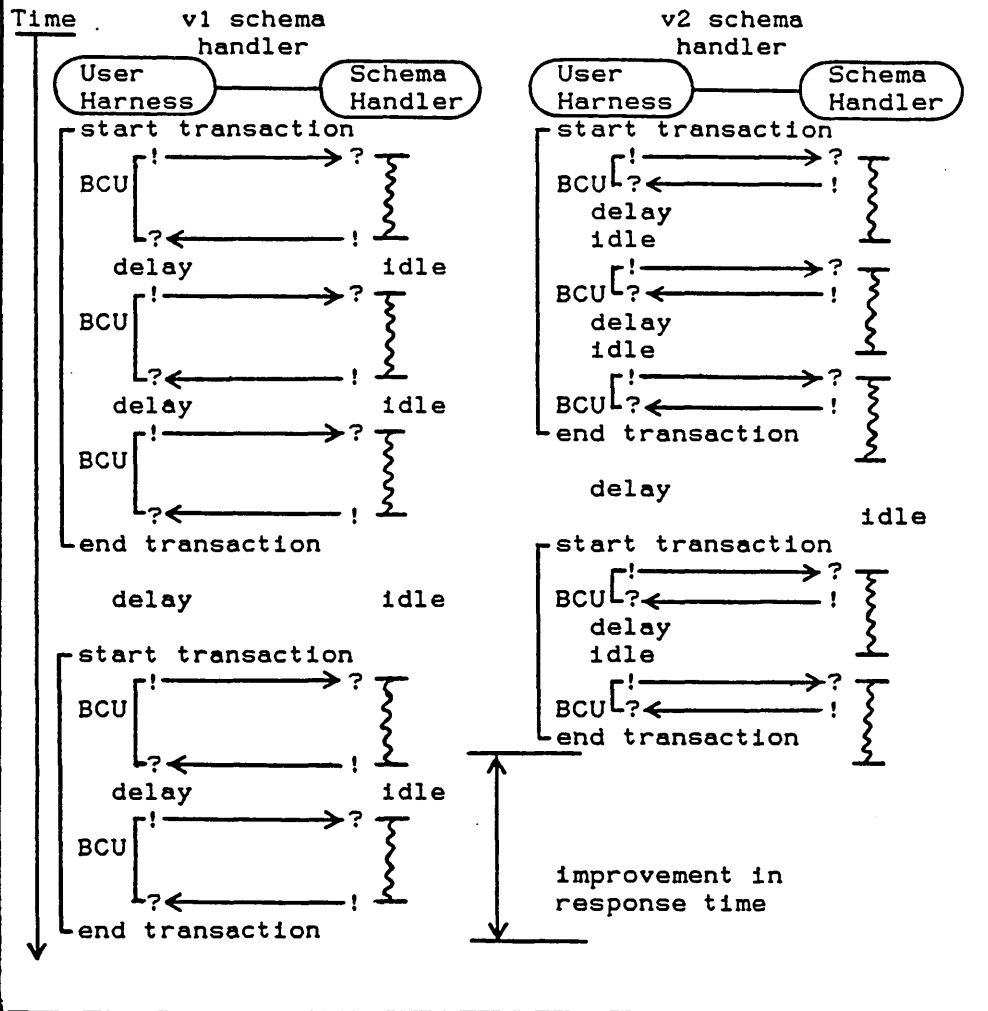


Figure F4 - Effect of User Harness Delays

Appendix G

Test Data and Test Configurations

This appendix lists the test data access schemas, the sets of test data run on these schemas, and the transputer network configurations used in the construction of the P-DB4GL test applications.

A data access schema is composed of prime view and non-prime view Information Units (IU's) and the schema links connecting related IU's. The test schemas are shown in Figure G2, and Figure G1 describes the IU's (or Test Entities) used in their composition. Each test entity has a name (ENTY.... for a prime-view IU, CPLE.... for a non-prime view IU) and a list of attributes. Attributes are identified by four digit numbers (0000-0019 Key, 0020-9999 non-key) and defined over domains. The type and length of the domain is indicated below the attribute number; 9 denotes numeric, and X denotes character, the length is in parantheses following the type.

Each test schema is implemented by a collection of data access processes, that is, a schema handler and a number of entity handlers; although SCHM0004 is slightly different, in that, it does not contain a schema handler, only prime and coupling entity handlers. Figure G3 illustrates the Data Access (DA) code used to implement the test schemas. The schema handlers and entity handler are connected by BCU channel pairs. The Filer and Disc processes are connected by channels conforming to the FILER protocols.

For each test schema there is a corresponding set of test data. A test data set consists of a sequence of transactions performed on the test schema; typically this involves Opening entity handlers, Storing records (IU occurrences), Reading and Deleting the stores records, followed by Closing and Terminating entity handlers. Each transaction is decomposed into a sequence of BCU request-reply messages. The User Process Test Harness (User Harness), operating in batch mode, supplies the test data as a sequence of BCU messages to the Data Access (DA) code. For those test schemas with schema links defined in them, some transactions will invoke schema link realisations; similarly, for test schemas with coupling entities (non-prime view IU's), some transactions will invoke coupling entity updates by prime entity handlers. Each test data set lists the number of transactions and BCU messages it contains and the number of schema link realisations and coupling entity updates invoked.

The test applications have been executed on several different transputer network configurations; these are illustrated in Figure G4. The most frequently used configurations are the single and two processor configurations. In the single processor configuration, both the User Harness and the Data Access (DA) code are executed as a TDS EXE process on the same processor. In the two processor configuration, the User Harness executes as a TDS EXE process, and the DA code (including the Filer and Disc processes) executes on a separate processor as a TDS PROG process. In order to obtain run time statistics of the DA code behaviour, Test Probe processes have normally been included in the test configurations. A Test Probe process, capable of interfacing to a terminal screen, executes on a separate processor (usually a T212-17 on a B006 board). Each Test Probe is connected via an Occam channel to one of the DA processes, and receives and displays run time diagnostic statistics generated internally by the DA process (for example, number of BCU's received, number of schema link realisations invoked, number of coupling entities updated). Also shown in Figure G4 is a four processor configuration (without test probes); this is the largest configuration used for the test applications.

Test Data Sets

Set 1

Run on test schema SCHM0001. Stores and reads 18 records in 3 prime entities. (6 for each entity). Consists of 39 transactions and 141 BCU request-reply message pairs. 54 schema link realisations are invoked. Sequence is:

- Open entities (1 transaction)
- Store 18 records (18 transactions)
- Read 18 records (18 transactions)
- Close entities (1 transaction)
- Terminate entities (1 transaction)

Set 2

Run on test schema SCHM0002. Stores and reads 12 records in 2 prime entities. (6 for each entity). Consists of 27 transactions and 90 BCU request-reply message pairs. 18 schema link realisations are invoked. Sequence is:

- Open entities (1 transaction)
- Store 12 records (12 transactions)
- Read 12 records (12 transactions)
- Close entities (1 transaction)
- Terminate entities (1 transaction)

Set 3

Run on test schema SCHM0003. Stores, reads, and deletes 18 records in 3 prime entities. (6 for each entity). Consists of 57 transactions and 177 BCU request-reply message pairs. 72 schema link realisations are invoked. Sequence is:

- Open entities (1 transaction)
- Store 18 records (18 transactions)
- Read 18 records (18 transactions)
- Delete 18 records (18 transactions)
- Close entities (1 transaction)
- Terminate entities (1 transaction)

Set 4

Run on test schema SCHM0004. Stores, reads, and deletes 6 records in a single prime entity. Consists of 21 transactions and 87 BCU request-reply message pairs. No schema link realisations are invoked (test schema SCHM0004 does not include a schema handler). The prime entity handler sends 59 BCU messages to each entity handler in the test schema. Sequence is:

- Open entity (1 transaction)
- Store 6 records (6 transactions)
- Read 6 records (6 transactions)
- Delete 6 records (6 transactions)
- Close entity (1 transaction)
- Terminate entity (1 transaction)

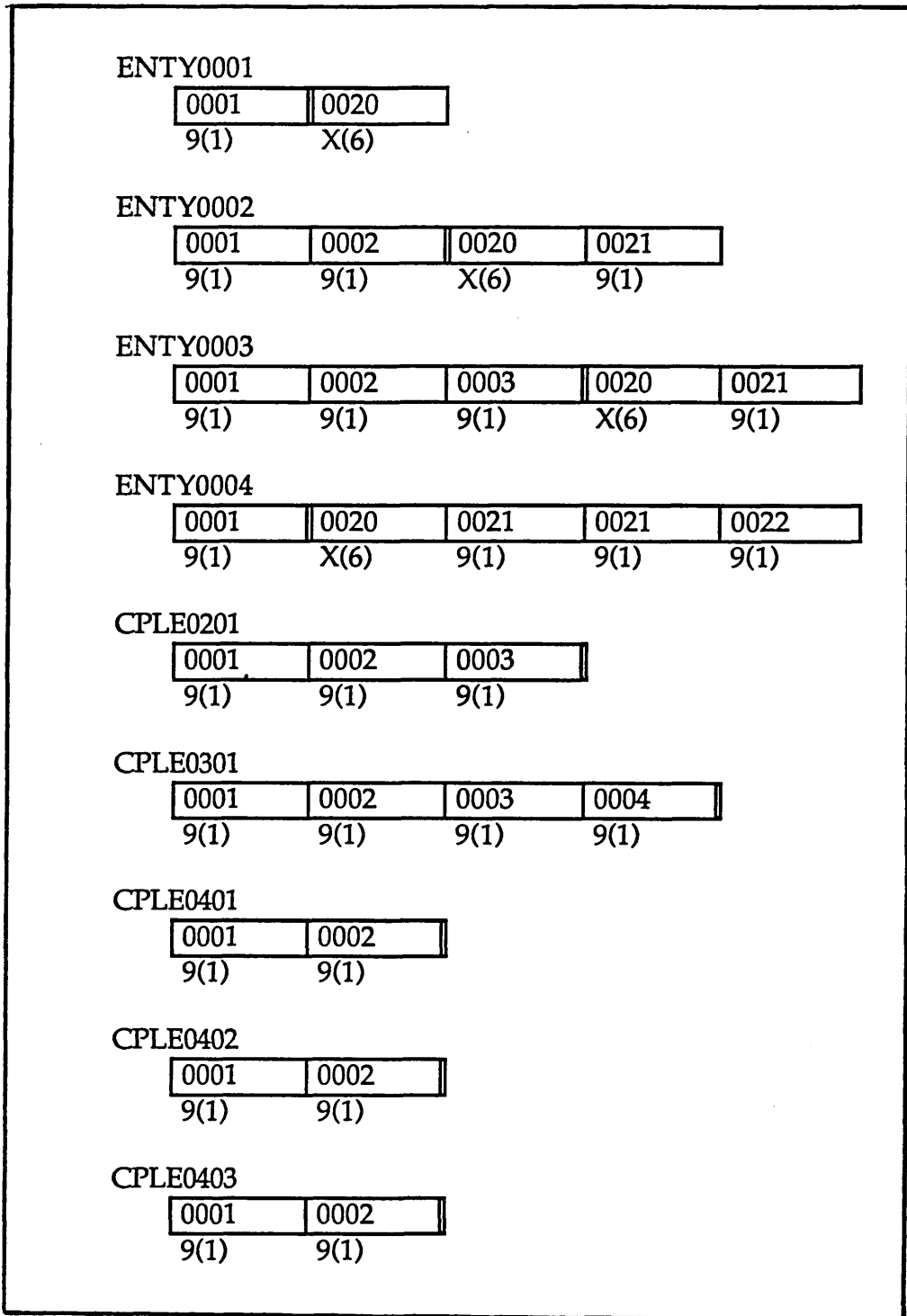


Figure G1 - Test Entities (Information Units)

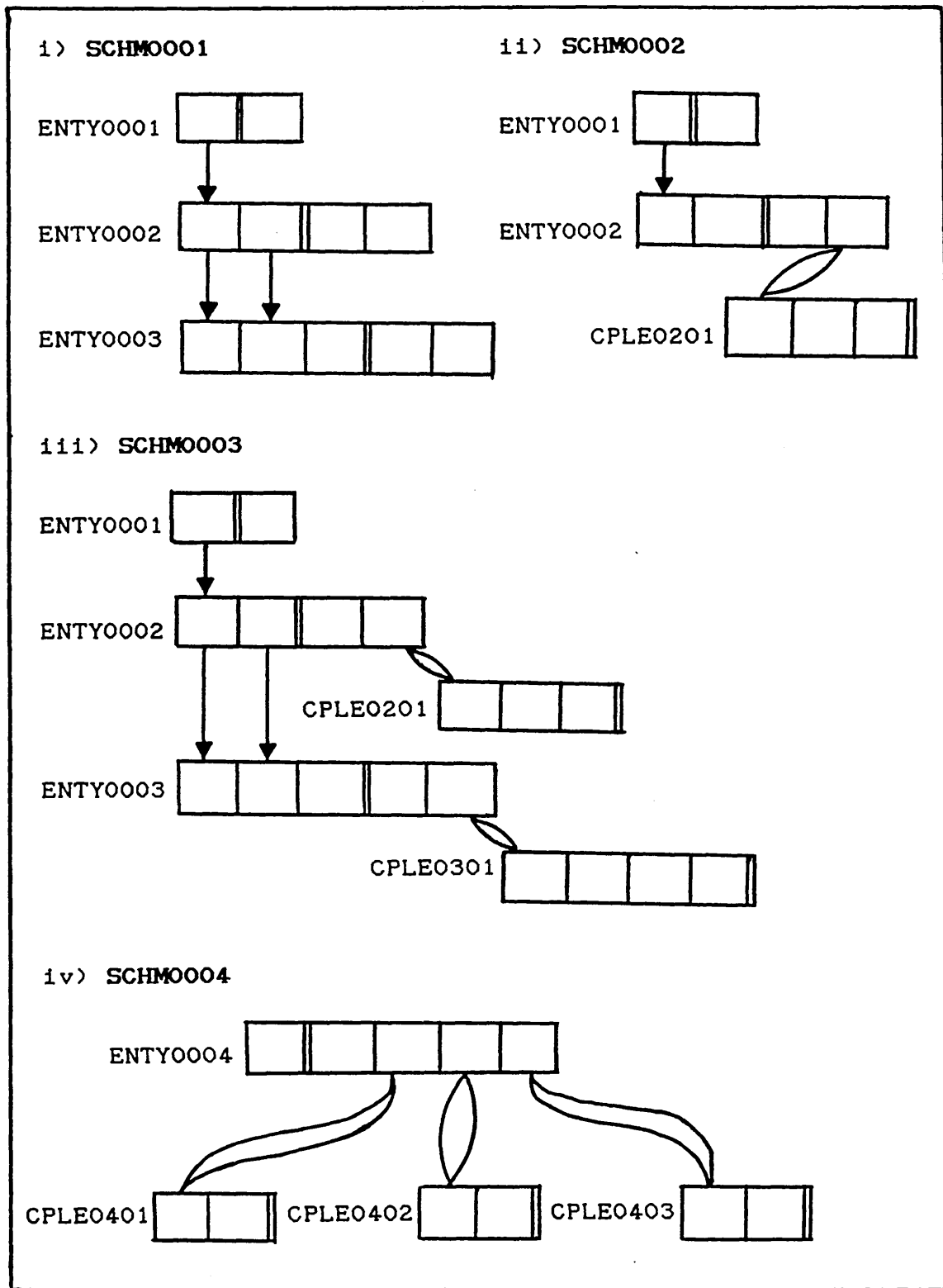


Figure G2 - Test Schemas

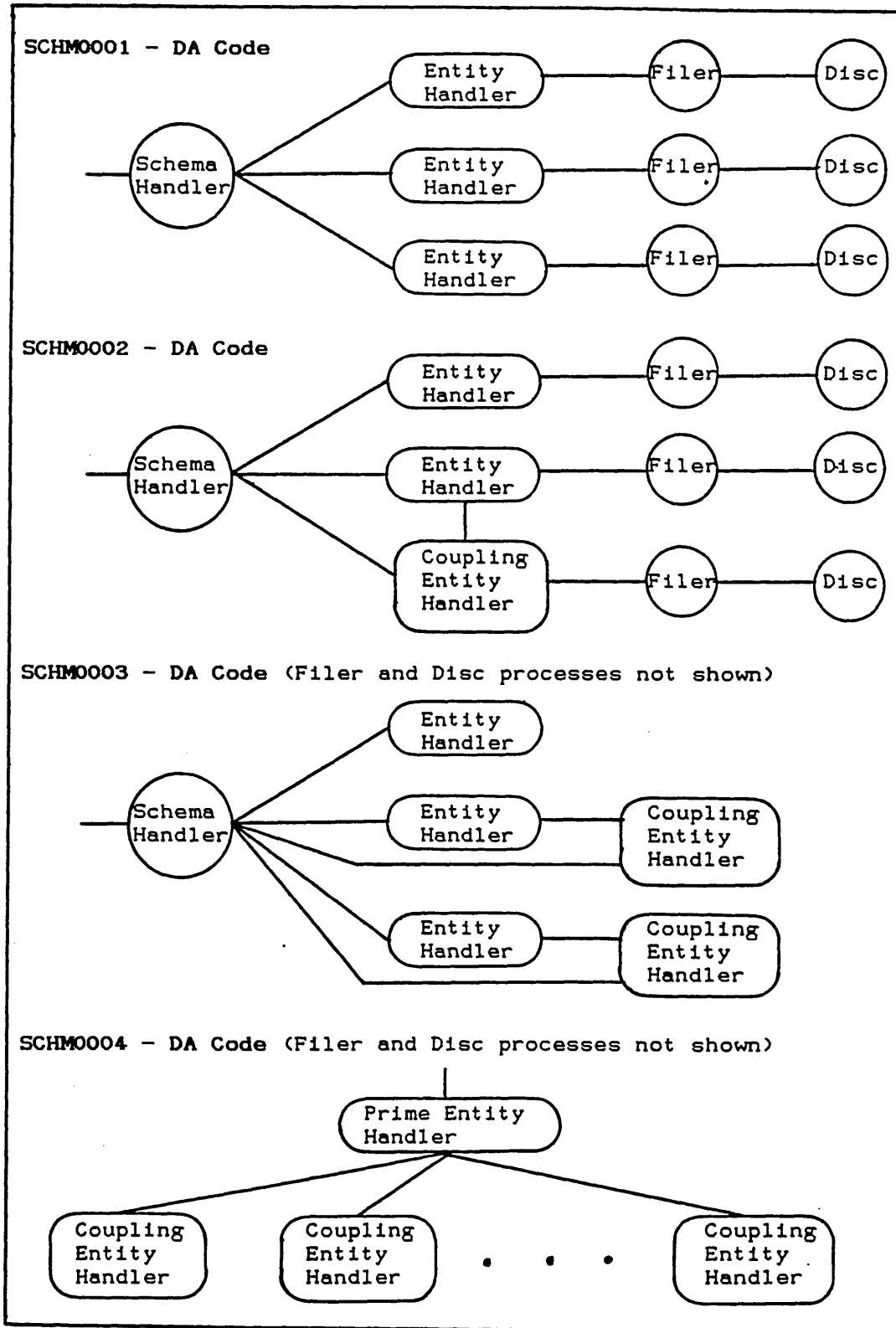


Figure G3 - Test Data Access Code

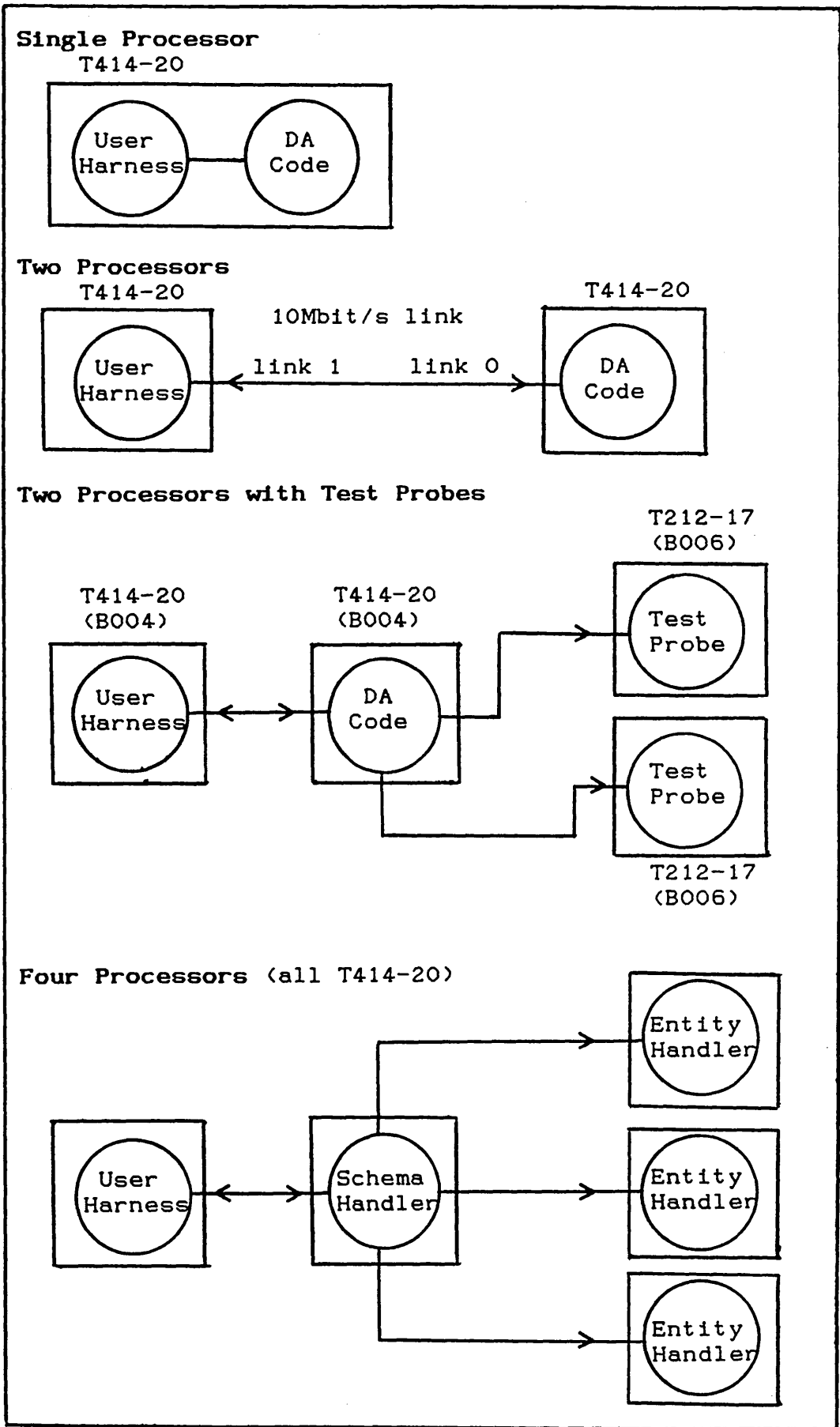


Figure G4 - Test Configurations

Appendix H

Test Results

This appendix contains the results obtained from test runs performed on the Prototype P-DB4GL Test Applications defined in Appendix G. The test run timings illustrate the differences in performance between the different versions of the data access processes (entity handlers and schema handlers) used in the construction of the test applications.

Tables 1-5 compare version 1 with version 2 data access processes. Test schemas SCHM0001, SCHM0002, and SCHM0003 are used, with varying User Harness delays and Filer Harness disc delays (Filer.v1 process).

Tables 6-11 compare version 1 with version 3 data access processes in SCHM0003. A variety of different disc delays and mappings of files to discs are used (Filer.v1 and Filer.v2 processes).

Tables 12-15 compare version 3 with version 4 data access processes using SCHM0004 and Filer.v2. The results show the effects on test run time and processor loading of the concurrent coupling entity update by version 4 prime entity handler. In particular, the significant improvement shown by increasingly larger test schemas (ie more coupling entities to update).

Table 1 Version 2 Schema Handler

A test run of version 1 entity and schema handlers, compared with a test run of version 1 entity handlers and version 2 schema handler.

Test data Set 1 run on test schema SCHM0001.

Two processor configuration, both T414-20.

Filestore simulation: Filer.v1 with one disc per file and 64 millisecond disc delay.

User Harness delays in timer ticks (one tick = 64 microseconds).

Test run time in seconds.

Inter-BCU delays (ticks)	0	50	200	500	1000	2000
Version 1 schema test run time	8.86	9.27	10.5	13.0	17.1	25.4
Version 2 schema test run time	8.85	9.14	10.0	11.8	14.8	21.1
% reduction in test run time	0	1.40	4.76	9.23	13.5	16.9
Improvement factor	1.00	1.01	1.05	1.10	1.16	1.20

Table 2 Version 2 Entity Handler

A test run of version 1 entity and schema handlers, compared with a test run of version 2 entity handlers and version 1 schema handler.

Test data Set 2 run on test schema SCHM0002.

Two processor configuration: User Harness, T414-20; DA processes, T414-15.

Filestore simulation: Filer.v1 with one disc per file and 64 millisecond disc delay.

User Harness delays in timer ticks (one tick = 64 microseconds).

Test run time in seconds.

Inter-BCU delays	0	0	0	200	500	1000
Inter-transaction delays	0	1000	2000	0	0	0
Version 1 entity test run time	4.88	6.61	8.33	6.03	7.76	10.64
Version 2 entity test run time	4.88	6.09	7.42	5.90	7.46	10.11
% reduction in test run time	0	7.87	10.92	2.16	3.87	4.98
Improvement factor	1.00	1.09	1.12	1.02	1.04	1.05
Inter-BCU delays	200	500	1000	500	1000	2000
Inter-transaction delays	1000	1000	1000	2000	2000	2000
Version 1 entity test run time	7.76	9.49	12.37	11.21	14.09	19.86
Version 2 entity test run time	7.15	8.76	11.45	10.28	13.16	18.92
% reduction in test run time	7.86	7.69	7.44	8.30	6.60	4.73
Improvement factor	1.09	1.08	1.08	1.09	1.07	1.05

Table 3 Version 2 Entity and Schema Handlers

A test run of version 1 entity and schema handlers, compared with a test run of version 2 entity and schema handlers.

Test data Set 2 run on test schema SCHM0002.

Two processor configuration, both T414-20.

Filestore simulation: Filer.v1 with one disc per file and 64 millisecond disc delay.

User Harness delays in timer ticks (one tick = 64 microseconds).

Test run time in seconds.

Inter-BCU delays	0	0	0	200	500	1000
Inter-transaction delays	0	1000	2000	0	0	0
Version 1 handlers test run time	4.81	6.54	8.26	5.73	7.11	9.41
Version 2 handlers test run time	4.81	6.02	7.35	5.61	6.82	8.90
% reduction in test run time	0	7.95	11.0	2.09	4.08	5.42
Improvement factor	1.00	1.09	1.12	1.02	1.04	1.06
Inter-BCU delays	200	500	1000	500	1000	2000
Inter-transaction delays	1000	1000	1000	2000	2000	2000
Version 1 handlers test run time	7.46	8.84	11.14	10.60	12.87	17.83
Version 2 handlers test run time	6.85	8.12	10.23	9.64	11.95	16.91
% reduction in test run time	8.18	8.14	8.17	9.06	7.15	5.16
Improvement factor	1.09	1.09	1.09	1.10	1.08	1.05

Table 4 Version 2 Handlers on a Larger Test Schema

A test run of version 1 entity and schema handlers, compared with a test run of version 2 entity and schema handlers.

Test data Set 3 run on test schema SCHM0003.

Two processor configuration, both T414-20.

Filestore simulation: Filer.v1 with one disc per file and 64 millisecond disc delay.

User Harness delays in timer ticks (one tick = 64 microseconds).

Test run time in seconds.

Inter-BCU delays	0	0	0	200	500	1000
Inter-transaction delays	0	1000	2000	0	0	0
Version 1 handlers test run time	16.10	19.75	23.39	18.36	21.76	27.43
Version 2 handlers test run time	13.91	17.09	20.66	15.33	17.67	21.69
% reduction in test run time	13.6	13.5	11.7	16.5	18.8	20.9
Improvement factor	1.16	1.16	1.13	1.20	1.23	1.26
Inter-BCU delays	200	500	1000	500	1000	2000
Inter-transaction delays	1000	1000	1000	2000	2000	2000
Version 1 handlers test run time	22.01	25.41	31.07	29.06	34.72	46.05
Version 2 handlers test run time	16.94	21.16	25.27	24.76	28.90	37.46
% reduction in test run time	23.0	16.7	18.7	14.8	16.8	18.7
Improvement factor	1.30	1.20	1.23	1.17	1.20	1.23

Table 5 Version 2 Handlers with a Shorter Disc Delay

A test run of version 1 entity and schema handlers, compared with a test run of version 2 entity and schema handlers.

Test data Set 3 run on test schema SCHM0003.

Two processor configuration, both T414-20.

Filestore simulation: Filer.v1 with one disc per file and 32 millisecond disc delay.

User Harness delays in timer ticks (one tick = 64 microseconds).

Test run time in seconds.

Inter-BCU delays	0	0	0	200	500	1000
Inter-transaction delays	0	1000	2000	0	0	0
Version 1 handlers test run time	8.48	12.13	15.78	10.75	14.15	19.81
Version 2 handlers test run time	7.35	10.71	14.31	8.81	11.22	15.44
% reduction in test run time	13.3	11.7	9.32	18.0	20.7	22.1
Improvement factor	1.15	1.13	1.10	1.22	1.26	1.28
Inter-BCU delays	200	500	1000	500	1000	2000
Inter-transaction delays	1000	1000	1000	2000	2000	2000
Version 1 handlers test run time	14.40	17.79	23.46	21.44	27.11	38.43
Version 2 handlers test run time	12.38	14.81	19.03	18.43	22.68	32.64
% reduction in test run time	14.0	16.8	18.9	14.0	16.3	15.1
Improvement factor	1.16	1.20	1.23	1.16	1.20	1.18

Table 6 Version 3 Entity and Schema Handlers

A test run of version 1 entity and schema handlers, compared with a test run of version 3 entity and schema handlers.

Test data Set 3 run on test schema SCHM0003.

Two processor configuration, both T414-20.

Filestore simulation: Filer.v1 with one disc per file and 32 millisecond disc delay.

User Harness delays in timer ticks (one tick = 64 microseconds).

Test run time in seconds.

Inter-BCU delays	0	0	0	200	500	1000
Inter-transaction delays	0	1000	2000	0	0	0
Version 1 handlers test run time	8.48	12.13	15.78	10.75	14.15	19.81
Version 3 handlers test run time	6.02	7.92	10.86	7.08	9.41	14.22
% reduction in test run time	29.0	34.7	31.2	34.1	33.5	28.2
Improvement factor	1.41	1.53	1.45	1.52	1.50	1.39
Inter-BCU delays	200	500	1000	500	1000	2000
Inter-transaction delays	1000	1000	1000	2000	2000	2000
Version 1 handlers test run time	14.40	17.79	23.46	21.44	27.11	38.43
Version 3 handlers test run time	9.54	12.37	17.49	15.75	21.13	32.64
% reduction in test run time	33.8	30.5	25.4	26.5	22.1	15.1
Improvement factor	1.51	1.43	1.34	1.36	1.28	1.18

Table 7 Version 3 Handlers on a Single Processor

A test run of version 1 entity and schema handlers, compared with a test run of version 3 entity and schema handlers.

Test data Set 3 run on test schema SCHM0003.

Single processor configuration, T414-20.

Filestore simulation: Filer.v1 with one disc per file and 32 millisecond disc delay.

User Harness delays in timer ticks (one tick = 64 microseconds).

Test run time in seconds.

Inter-BCU delays	0	0	0	200	500	1000
Inter-transaction delays	0	1000	2000	0	0	0
Version 1 handlers test run time	8.52	12.17	15.82	10.79	14.19	19.85
Version 3 handlers test run time	6.06	7.97	10.92	7.13	9.45	14.26
% reduction in test run time	28.9	34.5	31.0	33.9	33.4	28.2
Improvement factor	1.41	1.53	1.45	1.51	1.50	1.39
Inter-BCU delays	200	500	1000	500	1000	2000
Inter-transaction delays	1000	1000	1000	2000	2000	2000
Version 1 handlers test run time	14.43	17.83	23.50	21.48	27.15	38.47
Version 3 handlers test run time	9.58	12.41	17.54	15.79	21.18	32.51
% reduction in test run time	33.6	30.4	25.4	26.5	22.0	15.5
Improvement factor	1.51	1.44	1.34	1.36	1.28	1.18

Table 8 Version 3 Handlers with Improved Filer Process

A test run of version 1 entity and schema handlers, compared with a test run of version 3 entity and schema handlers.

Test data Set 3 run on test schema SCHM0003.

Two processor configuration, both T414-20.

Filestore simulation: Filer.v2 with one disc per file and 32 millisecond disc delay.

User Harness delays in timer ticks (one tick = 64 microseconds).

Test run time in seconds.

Inter-BCU delays	0	0	0	200	500	1000
Inter-transaction delays	0	1000	2000	0	0	0
Version 1 handlers test run time	6.00	9.65	13.30	8.27	11.67	17.33
Version 3 handlers test run time	4.27	6.49	9.96	5.30	8.00	13.45
% reduction in test run time	28.8	32.7	25.1	35.9	31.4	22.4
Improvement factor	1.41	1.49	1.34	1.56	1.46	1.29
Inter-BCU delays	200	500	1000	500	1000	2000
Inter-transaction delays	1000	1000	1000	2000	2000	2000
Version 1 handlers test run time	11.92	15.31	20.98	18.96	24.63	35.95
Version 3 handlers test run time	8.33	11.46	17.05	15.08	20.69	32.02
% reduction in test run time	30.1	25.1	18.7	20.5	16.0	10.9
Improvement factor	1.43	1.34	1.23	1.26	1.19	1.12

Table 9 Version 3 Handlers with Zero Disc Delay

A test run of version 1 entity and schema handlers, compared with a test run of version 3 entity and schema handlers.

Test data Set 3 run on test schema SCHM0003.

Two processor configuration, both T414-20.

Filestore simulation: Filer.v2 with one disc per file and 0 millisecond disc delay.

User Harness delays in timer ticks (one tick = 64 microseconds).

Test run time in seconds.

Inter-BCU delays	0	0	0	200	500	1000
Inter-transaction delays	0	1000	2000	0	0	0
Version 1 handlers test run time	0.883	4.53	8.18	3.15	6.55	12.21
Version 3 handlers test run time	0.913	4.13	7.76	2.58	5.93	11.56
% reduction in test run time	(3.4)	8.8	5.1	18.1	9.5	5.3
Improvement factor	0.97	1.10	1.05	1.22	1.10	1.06
Inter-BCU delays	200	500	1000	500	1000	2000
Inter-transaction delays	1000	1000	1000	2000	2000	2000
Version 1 handlers test run time	6.80	10.19	15.86	13.84	19.51	30.83
Version 3 handlers test run time	6.16	9.53	15.19	13.17	18.84	30.16
% reduction in test run time	9.4	6.5	4.2	4.8	3.4	2.2
Improvement factor	1.10	1.07	1.04	1.05	1.04	1.02

Table 10 Single Processor and Two Processor Configurations

Test runs of version 1 entity and schema handlers, compared with test runs of version 3 entity and schema handlers.

Test data Set 3 run on test schema SCHM0003.

Single processor configurations, T414-20.

Two processor configurations, both T414-20.

Filestore simulation: Filer.v2 with one disc per file - 0, 16, and 32 millisecond disc delays.

No User Harness delays.

Test run time in seconds.

Test run time for two processor configuration with Echo process replacing the test schema code is 0.042 880 seconds (approximately 5% of test schema run time).

Disc process delay (ms)	0		16		32	
Number of processors	One	Two	One	Two	One	Two
Version 1 handlers test run time	0.976	0.883	3.54	3.44	6.10	6.00
Version 3 handlers test run time	1.03	0.913	2.64	2.54	4.37	4.27
% reduction in test run time	(5.50)	(3.4)	25.4	26.2	28.4	28.8
Improvement factor	0.95	0.97	1.34	1.35	1.40	1.41

Table 11 Version 3 Handlers with a Single Disc

A test run of version 1 entity and schema handlers, compared with a test run of version 3 entity and schema handlers.

Test data Set 3 run on test schema SCHM0003.

Single processor configuration, T414-20.

Filestore simulation: Filer.v2 with all files stored on one disc, and 32 millisecond disc delay.

User Harness delays in timer ticks (one tick = 64 microseconds).

Test run time in seconds.

Inter-BCU delays	0	0	0	200	500	1000
Inter-transaction delays	0	1000	2000	0	0	0
Version 1 handlers test run time	6.06	9.71	13.35	8.82	11.72	17.39
Version 3 handlers test run time	5.77	7.18	10.54	6.32	8.29	13.59
% reduction in test run time	4.8	26.1	21.0	21.3	29.3	21.9
Improvement factor	1.05	1.35	1.27	1.40	1.41	1.28
Inter-BCU delays	200	500	1000	500	1000	2000
Inter-transaction delays	1000	1000	1000	2000	2000	2000
Version 1 handlers test run time	11.97	15.37	21.03	19.02	24.68	36.01
Version 3 handlers test run time	8.92	11.68	17.18	15.30	20.83	32.07
% reduction in test run time	25.5	24.0	18.3	17.7	15.6	10.9
Improvement factor	1.34	1.32	1.22	1.24	1.18	1.12

Table 12 Version 4 Entity Handlers

A test run of version 3 entity handlers, compared with a test run of version 4 prime and coupling entity handlers.

Test data Set 4 run on test schema SCHM0004 (three coupling entities, no schema handler).

Two processor configuration, both T414-20.

Filestore simulation: Filer.v2 with one disc per file and 32 millisecond disc delay.

User Harness delays in timer ticks (one tick = 64 microseconds).

Test run time in seconds.

Inter-BCU delays	0	0	0	200	500	1000
Inter-transaction delays	0	1000	2000	0	0	0
Version 3 handlers test run time	2.42	2.93	3.68	3.36	4.78	7.15
Version 4 handlers test run time	1.41	2.14	3.45	2.35	3.77	6.37
% reduction in test run time	41.7	27.0	6.3	30.1	21.1	10.9
Improvement factor	1.72	1.37	1.07	1.43	1.27	1.12
Inter-BCU delays	200	500	1000	500	1000	2000
Inter-transaction delays	1000	1000	1000	2000	2000	2000
Version 3 handlers test run time	3.87	5.29	7.90	6.37	9.06	14.59
Version 4 handlers test run time	3.22	4.89	7.68	6.24	9.02	14.59
% reduction in test run time	20.2	7.6	2.8	2.0	0.4	0
Improvement factor	1.20	1.08	1.03	1.02	1.004	1.00

Table 13 Version 4 Handlers with Varying Numbers of Coupling Entities

Test runs of version 3 entity handlers, compared with test runs of version 4 prime and coupling entity handlers.

Test data Set 4 run on test schema SCHM0004 (with between 0 and 18 coupling entities, no schema handler).

Two processor configuration, both T414-20.

Filestore simulation: Filer.v2 with one disc per file and 64 millisecond disc delay.

No User Harness delays.

Test run time in seconds.

Number of coupling entities updated	0	1	3	6	9	12	15	18
Version 3 handlers test run time	1.13	2.53	4.66	7.85	11.03	14.21	17.40	20.58
Version 4 handlers test run time	1.14	2.55	2.63	2.74	2.86	2.99	3.13	3.27
Test run time improvement factor	0.99	0.99	1.77	2.86	3.86	4.75	5.56	6.29
Version 3 handlers coupling update time	-	1.40	3.53	6.72	9.90	13.08	16.27	19.45
Version 4 handlers coupling update time	-	1.41	1.49	1.60	1.72	1.85	1.99	2.13
Version 3 normalised coupling update time	-	1.00	2.52	4.80	7.07	9.34	11.62	13.89
Version 4 normalised coupling update time	-	1.01	1.06	1.14	1.23	1.32	1.42	1.52
Coupling update improvement factor	-	0.99	2.38	4.21	5.75	7.08	8.18	9.14

Table 14 Version 4 Handlers with Zero Disc Delay

Test runs of version 3 entity handlers, compared with test runs of version 4 prime and coupling entity handlers.

Test data Set 4 run on test schema SCHM0004 (with between 0 and 18 coupling entities, no schema handler).

Two processor configuration, both T414-20.

Filestore simulation: Filer.v2 with one disc per file and 0 millisecond disc delay.

No User Harness delays.

Test run time in milliseconds.

Number of coupling entities updated	0	1	3	6	9	12	15	18
Version 3 handlers test run time	45.6	102.8	180.4	294.8	407.8	518.8	628.9	741.8
Version 4 handlers test run time	54.6	116.5	207.5	348.1	495.4	647.9	808.0	975.8
Version 3 handlers coupling update time	-	57.2	134.8	249.2	362.5	473.2	583.3	696.2
Version 4 handlers coupling update time	-	61.9	152.9	293.5	440.8	593.3	753.4	921.2
Version 3 normalised coupling update time	-	1.00	2.36	4.36	6.34	8.27	10.20	12.17
Version 4 normalised coupling update time	-	1.08	2.67	5.13	7.71	10.37	13.17	16.10

Table 15 Test Application Processor Loading

Test runs of version 3 entity handlers, compared with test runs of version 4 prime and coupling entity handlers. Showing the percentage of total test run time during which the processor is busy (see Tables 13 and 14).

$$\text{Processor Loading} = \frac{\text{test run time (0 disc delay)}}{\text{test run time (64 ms disc delay)}} * 100\%$$

Test data Set 4 run on test schema SCHM0004 (with between 0 and 18 coupling entities, no schema handler).

Two processor configuration, both T414-20.

Filestore simulation: Filer.v2 with one disc per file.

No User Harness delays.

Number of coupling entities updated	0	1	3	6	9	12	15	18
Version 3 handlers processor loading %	4.04	4.06	3.87	3.76	3.70	3.65	3.61	3.60
Version 4 handlers processor loading %	4.79	4.57	7.89	12.70	17.32	21.67	25.80	29.80

Appendix I

Test Application Performance Optimization

There are many ways of optimizing the performance of Occam programs executing on transputer networks. Small alterations to both the software and hardware, which do not affect the logical behaviour of the program, can have a marked effect on the execution times of many programs [Atkin89]. These optimizations cannot be relied upon to improve execution times of all programs, in fact, for some programs an "optimization" may actually impair performance. It is therefore necessary to experiment with these optimizations to determine their effect on any given program. A list of such optimizations applied to the P-DB4GL applications, along with comments concerning the possible (and sometimes measured) effect on test application performance, is given below.

Processor Type and Speed

The transputer range of microprocessors includes both 16-bit and 32-bit processors with different features and performance characteristics. The T800 32 bit processor has an on-chip floating point unit (FPU) which allows a T800 to perform floating point calculations up to fifty times faster than a 32-bit T414. However, not all operations are faster on a 32-bit processor than a 16-bit processor, the T212 performs 16-bit (INT16) integer arithmetic faster than a 32-bit T414.

Any performance improvement that might be obtained by substituting one transputer processor with another of a different type, depends on the nature and number of the machine operations performed in a program. In the P-DB4GL test applications there is almost no floating point arithmetic, but substantial amount of INT16 arithmetic. Most of the P-DB4GL test runs have been on T414 processors, the few that have been on T800 processors exhibit very little difference in execution times.

The transputer microprocessors are available in various speeds, ranging from 15 to 30 MHz. Replacing a processor with another, but faster, processor of the same type will give an improvement in execution times. But the exact improvement depends on the proportion of processing and communication performed in the program. Programs that are

communication bound will not be improved simply by the substitution of faster processors.

Memory Speed and Interface

For the 32-bit processors there are two types of external memory interface with different memory bandwidths: a multiplexed address/data bus used on the T414 and T800 (40 Mbyte/s bandwidth); and a non-multiplexed interface with separate data and address lines used on the T801 (60 Mbyte/s bandwidth). The faster external memory bandwidth of the T801 can improve the performance of programs which require fast external memory access. However, the T801 is not pin compatible with either the T414 or T800 and a straightforward processor swap is not possible, it requires a different board.

For similar reasons the type and speed of external RAM can also affect the speed of program execution. But for any given program the precise benefit of either improved memory speed or memory bandwidth depends on the patterns of external memory usage during program execution and the overall proportions of processing and communication loads. P-DB4GL test applications tend to have high communication loads and the potential benefit of improved memory access is not great.

Link Speed and Protocol

Transputer links can be set to operate at either 5, 10 or 20 Mbit per second transfer rates. Increasing the link speeds in a network from say 10 to 20 Mbit per second will reduce the communication time for each message sent over these links; but the overall effect of this reduction in communication time depends on the amount and type of communication in the program executing on the network. Two problems in particular can be remedied by improved link speed:

- a program suffering from overall high levels of inter-processor communication (that is, communication bound) can be improved;
- a program with overall low levels of communication, but suffering from a "bottleneck" in communication at a particular inter-processor channel in the program, can be improved if the link speed at the "bottleneck" channel is increased.

There are two compatible versions of the handshaking protocol used on the transputer serial links [Inmos89a]. In the original protocol, a two bit acknowledge message is sent from the receiving end of a link when a complete data message has been received. In the modified protocol the acknowledge is overlapped with the data message. The modified protocol is used in the T800, T222, and T425 and gives an effective data transfer rate approximately twice as great as the original protocol used on the T414 and T212.

The P-DB4GL test applications have been executed on transputer networks constructed mostly from T414 processors with 10 Mbit/sec links, and the boards on which the processors are mounted have not permitted the link speeds to be altered. Because the P-DB4GL test applications perform large amounts of communication, it is anticipated that increasing link speeds to 20 Mbit/sec and replacing the T414 processors with T425 or T800 processors would improve execution times of the test applications. Such changes to the networks would increase maximum unidirectional data transfer rates from 400 Kbyte/sec to 1740 Kbyte/sec.

Program Configuration

Load balancing, that is, allocating processes to processors to achieve an even distribution of processing load across a network, can have a marked effect on program performance. Each concurrent process in an Occam program will have its own processing and communication load, and it is often necessary to experiment with alternative configurations, or mappings of code to processors, in order to find one with optimum balance and performance. Processes connected by channels with high communication loads are often best placed on the same processor, so that the channel communication is performed in memory rather than across links.

The P-DB4GL test applications have been executed on small transputer networks (one to four processors). Because of the I/O delays associated with retrieving records from disks, the processing loads of the test applications are relatively light in proportion to the total test run times. Consequently, altering the configuration of test applications for effective load balancing is not necessary. However, some channels do carry very large amounts of communication, and considerations of channel placement (in memory or on links) is important in the designs for a fully functional P-DB4GL.

Channel Protocols and Process Priority

The type of Occam protocol defined for channel communication can affect the data transfer rate when a channel is placed on a serial link (rather than in memory). Increasing a channel's data transfer rate will reduce the communication time for each message sent on that channel. The maximum effective data transfer rate across transputer links, as stated in the databooks [Inmos89b], can only be achieved when large messages are communicated over links. For example, the data transfer rate when four bytes of data are transmitted as a single message using a [4]BYTE array protocol is greater than if the same data is transmitted as a single message of four separate bytes using a protocol such as BYTE; BYTE; BYTE; BYTE.

Figure I1 shows the effect of altering message size (by using different Occam channel protocols) on the data transfer rate through an Occam channel placed on a transputer serial link. The data for this graph was obtained by running a small test program on a T800-20 connected to a C011 link adapter. The T800 test program executed entirely in the on-chip memory and consisted of a simple WHILE loop sending fixed size messages on the link adapter channel as fast as the link would take them. The transfer rates were measured for messages of different sizes, including single byte messages (BYTE protocol) and byte array messages of various sizes ([size]BYTE protocols). These results are for a link speed of 20 Mbit/s. The C011 link adapter uses the old (non-overlapped acknowledge) 2-bit handshake protocol. It can be seen that with large messages, that is byte arrays of size 64 and greater, the data transfer rate approaches its maximum value. This type of graph is typical for channel communication over links for the T-range of transputers.

The improved transfer rate, and consequent shorter communication times, of more efficient Occam channel protocols can be employed to improve the performance of many programs. However, it is necessary to experiment with different protocols to determine how much, if any, improvement can be obtained for any given program. For some programs such changes to the channel protocols may require substantial alterations to the source code. The time and cost of altering the source code, and the possibility of introducing new errors, may not justify any potential performance gains.

One way to alter the Occam protocol used on a link, without altering a program's source code, is to introduce protocol converter processes to execute concurrently with the existing program (Figure I2). This approach was tried for some of the P-DB4GL test applications. Figure I3 shows the configuration used: an Echo process executing on a T414 processor returns BCU-reply messages to a User Harness supplying a Test Data Set of BCU-request messages. Each BCU message is composed of several data and control components represented by BYTE and []BYTE array values. A protocol converter process translates this representation into a counted array protocol, BYTE::[]BYTE. The counted array protocol can attain higher data transfer rates across the connecting transputer serial link. Table A shows the effect on communication times of the protocol converter processes. It can be seen from these results that the protocol converters do not reduce the total test run times, in fact, they are slightly worse. Although the counted array protocol reduces the communication time for each BCU message, the additional processing overhead of the converter process combined with the Request-Reply nature of the BCU messages, which inhibits potential benefits of buffering and processing-communication decoupling, causes the total test run time to be increased.

The Occam language [Inmos88b] contains a PRI PAR construct which can be used to specify preferential, or priority, status to selected processes in a parallel PAR construct. There is also a PRI ALT construct which can be similarly used to allow preferential input of data on selected channels. The effect of using these two constructs in a program to enhance performance is not predictable. By experimenting with different process and channel priorities a program's execution time may be improved. In programs with high levels of inter-processor communication, giving priority status to processes communication over links can often reduce communication delays and hence reduce overall program execution times. To measure the effect of process priority upon communication times in the P-DB4GL test applications, the test configuration described above (Figure 3) was altered to give priority status to the protocol converter processes. The results of this alteration are shown in Table A: where it can be seen that this prioritization has a slightly detrimental effect on test run time.

Data Placement and Compiler Code Allocation

The Occam2 language includes a PLACE statement to allocate particular objects (variables, channels, timers, arrays) to specific memory locations. This can be used to place frequently accessed vectors or code fragments in a processor's internal memory thereby permitting faster execution. Again the precise effect of such changes is not predictable and can only be determined by experimentation.

The Occam2 compiler performs many optimizations on the generated machine code [Inmos88d], and knowing how the compiler works allows one to assist the compiler in this optimization. In particular, a knowledge of how the compiler allocates code and data to memory can be very useful. This allocation is in part determined by the position of statements in the Occam source code. By repositioning process definitions and variable declarations within the text of the source code (and taking care not to alter the semantics of the program) improvements in execution times can sometimes be obtained.

The effects of the PLACE statement and repositioning objects in the source code have not been tested on the P-DB4GL test applications. Although slight improvements may be gained, the experimentation entailed can be very time consuming.

Table A Optimizations to Test Application Communication Times

Test data Set 4 with Echo process replacing test schema code.
 Two processor configuration, both T414-20.
 User Harness delays in timer ticks (1 tick = 64 microseconds).
 Test run time in milliseconds.

Inter-BCU delays	0	0	0	200	500	1000
Inter-transaction delays	0	1000	2000	0	0	0
Test run time	20.22	1 364	2 708	1 134	2 804	5 588
Test run time with protocol conversion	27.14	1 371	2 715	1 140	2 811	5 595
Test run time with protocol conversion at high priority	31.23	1 375	2 719	1 145	2 815	5 599
Inter-BCU delays	200	500	1000	500	1000	2000
Inter-transaction delays	1000	1000	1000	2000	2000	2000
Test run time	2 478	4 148	6 932	5 492	6 932	13 844
Test run time with protocol conversion	2 484	4 155	6 939	5 499	8 283	13 851
Test run time with protocol conversion at high priority	2 489	4 159	6 943	5 503	8 287	13 855

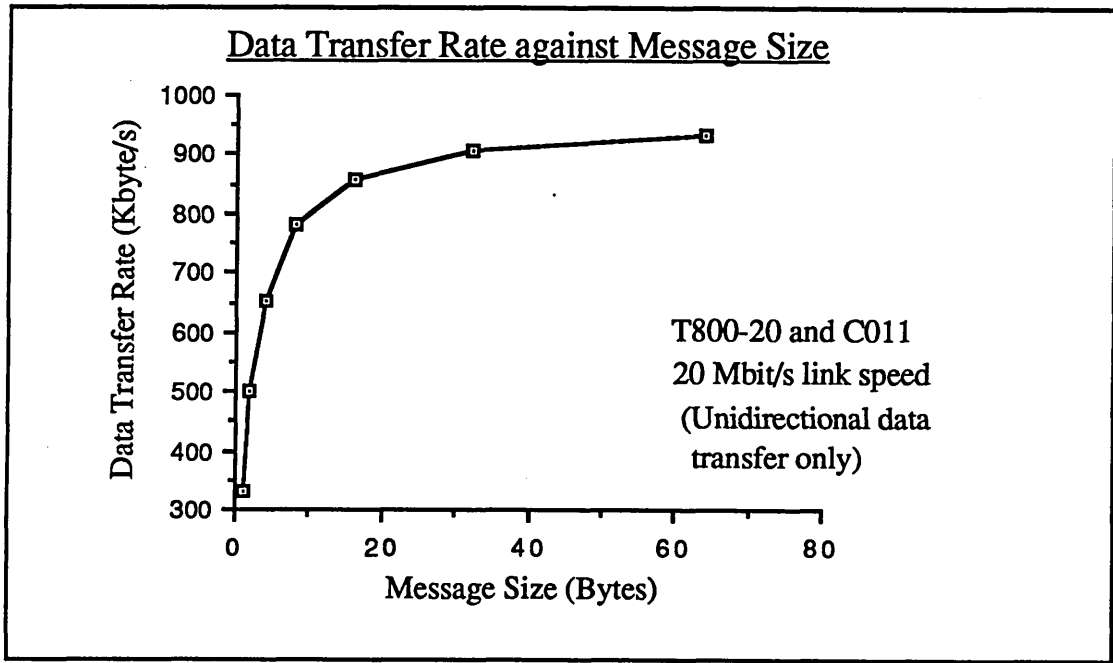


Figure I1 - Effect of Message Size on Transfer Rate

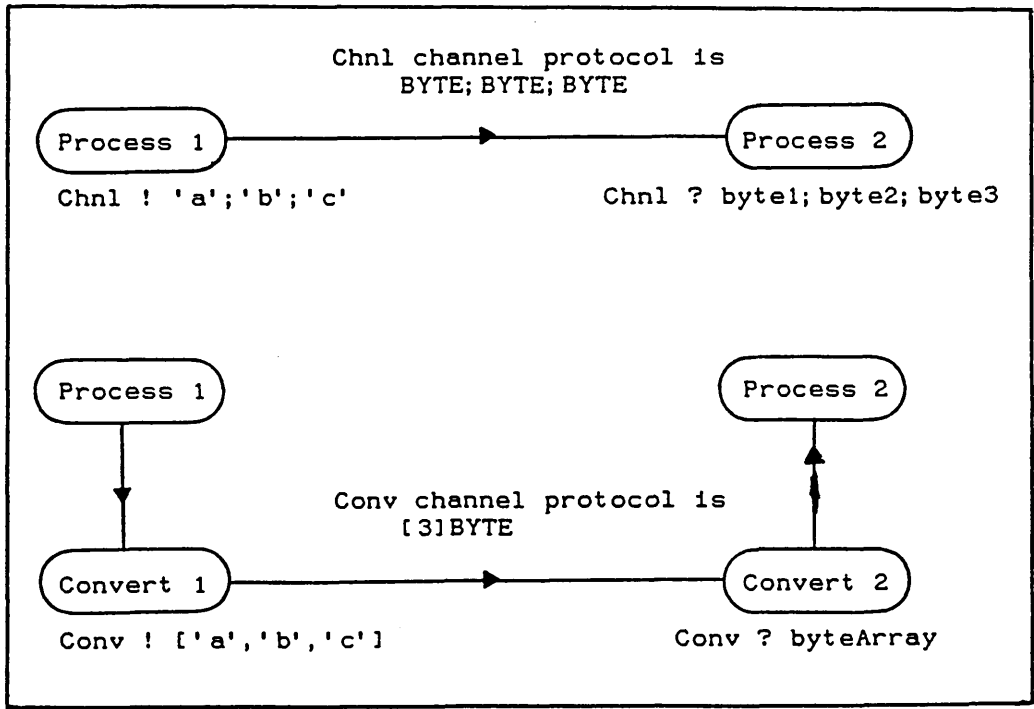


Figure I2 - Channel Protocol Converter Processes

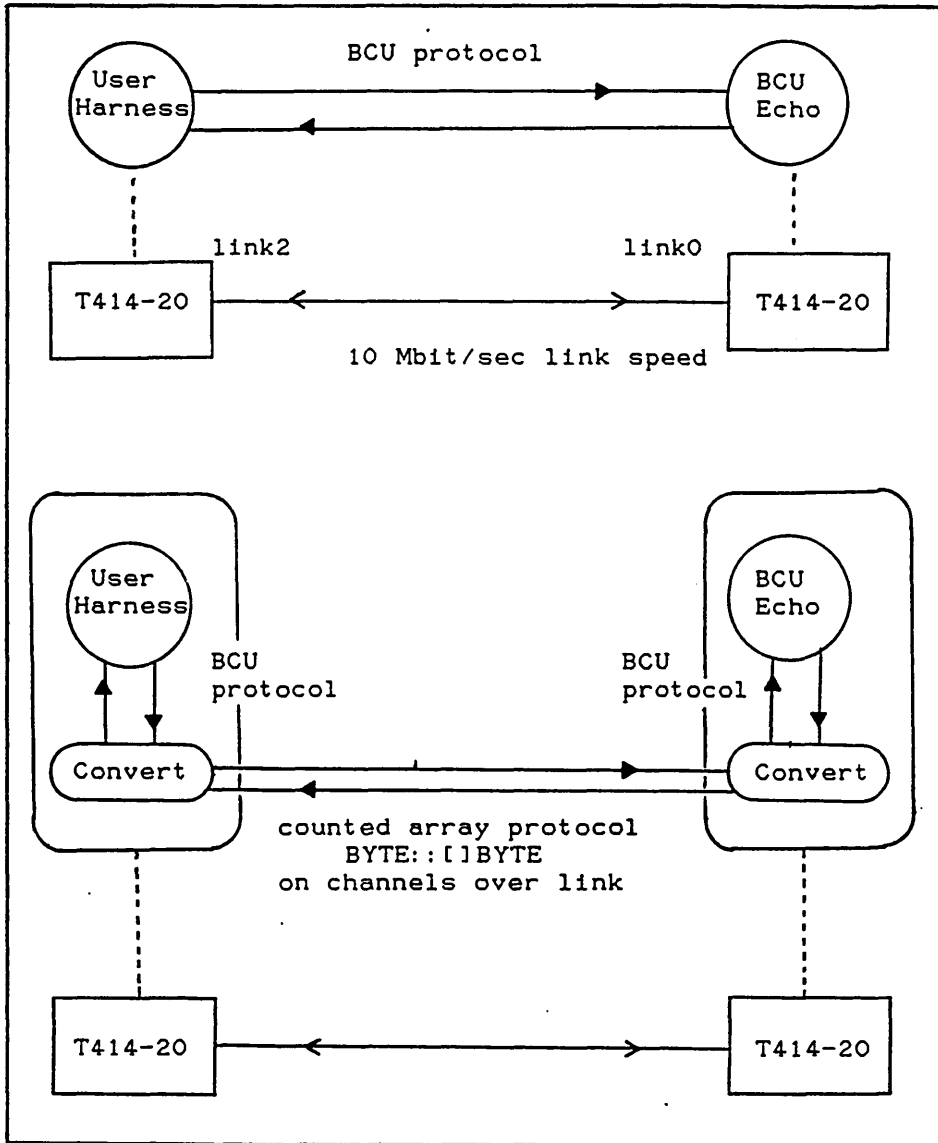


Figure I3 - Protocol Conversion Test Configuration

Appendix J

Routers and Multiplexors

Algorithms for the P-DB4GL code used to multiplex Occam channels and route messages around the transputer networks are described below using the Occam-style syntax defined in Appendix A. The data access processes used in the prototype P-DB4GL test applications have been modified with the addition of channel conversion and multiplexing processes. Two of these modified data access processes are: the "ENTY.handler" process, a modified prime entity handler; and the "CPL handler", a modified coupling entity handler.

```

PROC ENTY.handler(CHAN OF BCUX handler.in, handler.out,
                  CHAN OF FILER.REQ filer.req,
                  CHAN OF FILER.REPLY filer.reply,
                  CHAN OF ANY debug.out,
                  CHAN OF BYTE stopper.out,
                  ... VALue parameters
                  )
... eh.local.val.decs
... include libraries of procedures
-- local channels declared
CHAN OF BCUX couple.req, schm.req :
CHAN OF BCUX couple.reply, schm.reply :
CHAN OF BCU enty.schm.req, enty.schm.reply :
[max.no.of.couples]CHAN OF BCU enty.couple.req, enty.couple.reply :
CHAN OF BYTE stop.schm, stop.couple, stop.mux :
PAR
  -- prime entity handler with many BCU channels
  VAL BYTE stop IS 00(BYTE) :
  SEQ
    PrimeXZ (enty.schm.req,enty.schm.reply,
             enty.couple.reply,enty.couple.req,
             filer.req,
             filer.reply,
             debug.out,
             object.id,
             ... value parameters to initialise entity handler
             )
    -- terminate channel converters and multiplexor
    stop.schm ! stop
    stop.couple ! stop
    stop.mux ! stop
    -- inform decoder of handler termination
    stopper.out ! stop
  -- couple channel converter
  Couple.BCUX.convert(couple.reply, couple.req,
                     enty.couple.req, enty.couple.reply,
                     object.id,
                     couple.list,
                     no.of.couple.entities,
                     stop.couple)
  -- schema channel converter
  Schema.BCUX.convert(schm.req, schm.reply,
                     enty.schm.reply, enty.schm.req,
                     object.id,
                     stop.schm)
  -- channel mux
  BCUX.mux(handler.in, handler.out,
           couple.req, couple.reply,
           schm.reply, schm.req,
           stop.mux)

```

:

```

PROC CPLE.handler(CHAN OF BCUX handler.in, handler.out,
                  CHAN OF FILER.REQ filer.req,
                  CHAN OF FILER.REPLY filer.reply,
                  CHAN OF ANY debug.out,
                  CHAN OF BYTE stopper.out,
                  ... VALUE parameters
                  )
... eh.local.val.decs
... include libraries of procedures
-- local channels declared
CHAN OF BCUX prime.req, schm.req :
CHAN OF BCUX prime.reply, schm.reply :
CHAN OF BCU cple.prime.req, cple.schm.req :
CHAN OF BCU cple.prime.reply, cple.schm.reply :
CHAN OF BYTE stop.schm, stop.prime, stop.mux :
PAR
-- coupling entity handler with many BCU channels
VAL BYTE stop IS 00(BYTE) :
SEQ
  CoupleXZ(cple.prime.req, cple.prime.reply,
           cple.schm.req, cple.schm.reply,
           filer.req,
           filer.reply,
           debug.out,
           object.id,
           ... value parameters to initialise entity handler
           )
-- terminate channel converters and multiplexor
stop.schm ! stop
stop.prime ! stop
stop.mux ! stop
-- inform decoder of handler termination
stopper.out ! stop
-- prime channel converter
Prime.BCUX.convert(prime.req, prime.reply,
                  cple.prime.reply, cple.prime.req,
                  object.id,
                  prime.id,
                  stop.prime)
-- schema channel converter
Schema.BCUX.convert(schm.req, schm.reply,
                  cple.schm.reply, cple.schm.req,
                  object.id,
                  stop.schm)
-- channel mux
BCUX.mux(handler.in, handler.out,
         prime.reply, prime.req,
         schm.reply, schm.req,
         object.id,
         stop.mux)
:

```

The P-DB4GL Router process is decomposed into two concurrent components: a BCUXR Converter process; and a Network process (Figure J1). The Converter process includes a look-up table containing the node identifiers of all the P-DB4GL objects in a particular P-DB4GL application, and uses this table to encode incoming BCUX messages with the node identifier of the destination object identified in the message. The encoded BCUX message is forwarded as a BCUXR message to the Network process, which then routes the BCUXR message around the ring to the destination node. The BCUXR protocol is a BYTE::[]BYTE counted array protocol, where the first BYTE value is the length of the message contained in the following array. The zeroth element of the array contains the source node identifier, and the first element contains the destination node identifier.

The Network8 (Figure J2) and Network9 (Figure J3) ring routing processes are based on the ring routing processes described in [Welch89]. Essentially, these processes implement a slotted-ring composed of N packets, where N is the number of nodes in the ring. The ring of packets continually circulates around the ring of processors. Each node has its "own" packet in which it may place messages of varying size. When a message arrives at its destination node, it is taken off the ring and the packet marked as empty. The Network8 router process includes a Buffer process, which is used to ensure that the ring of packets will continue to circulate, even when a destination node is blocked and unable to accept further messages from the ring.

The Network8 process implements a unidirectional ring. The Network9 process uses two Network8 processes, along with Sender and Collect processes, to implement a bidirectional ring. The Sender process routes BCUXR messages either clockwise or anticlockwise, depending on which direction provides the shortest path to the destination node.

```

PROC Network8(CHAN OF BCUXR c.net.in, c.net.out,
              c.from.app, c.to.app,
              VAL BYTE this.node, no.of.nodes)
-- "own packet" ring algorithm
-- with N size message buffer
-- ring must continue to circulate even with a blocked node
VAL INT max.buffer.size IS 8 :
PROC Bcuxr.Buffer(CHAN OF BCUXR data.in, data.out)
  ... buffer body
:
PROC Tail.Buffer(CHAN OF BCUXR data.in, data.out,
                CHAN OF INT dec.counter)
  -- decrease counter by 1 as messages leave the buffer
  ... tail body
:
PROC Head.Buffer(CHAN OF BCUXR data.in, data.out,
                CHAN OF BOOL free.out,
                CHAN OF INT dec.counter,
                VAL INT no.of.nodes)
  -- when buffer has filled, flush out no.of.nodes messages
  -- before accepting any more messages into buffer.
  -- if no.of.nodes > max.buffer.size,
  -- then flush out max.buffer.size messages
  ... head body
:
PROC N.Size.Buffer(CHAN OF BCUXR data.in, data.out,
                  CHAN OF BOOL free.out,
                  VAL BYTE no.of.nodes)
  VAL INT node.count IS (INT no.of.nodes) :
  CHAN OF INT dec.counter :
  [max.buffer.size]CHAN OF BCUXR data.thru :
  PAR
    Head.Buffer(data.in, data.thru[0],
                free.out, dec.counter, node.count)
    PAR i = 1 FOR (max.buffer.size - 1)
      Bcuxr.Buffer(data.thru[i-1], data.thru[i])
    Tail.Buffer(data.thru[(max.buffer.size - 1)],
                data.out,dec.counter)
:
PROC Transmit(CHAN OF BCUXR app.in, thru.in, net.out,
              VAL BYTE this.node, no.of.nodes)
  -- accept messages from app and deposit in "own" packet
  ... transmit body
:
PROC Receive(CHAN OF BCUXR net.in, thru.out, app.out,
             CHAN OF BOOL free.in,
             VAL BYTE this.node )
  -- if buffer not free to accept message for this node,
  -- then force message to circulate around the ring again
  ... receive body
:
CHAN OF BCUXR clock.thru, rec.to.buff :
CHAN OF BOOL free.buffer :
PAR
  N.Size.Buffer(rec.to.buff, c.to.app, free.buffer, no.of.nodes)
  Receive(c.net.in, clock.thru, rec.to.buff, free.buffer, this.node)
  Transmit(c.from.app, clock.thru, c.net.out,
            this.node, no.of.nodes)
:

```



```

PROC Network9 (CHAN OF BCUXR from.app, to.app,
               c.net.in, c.net.out,
               a.net.in, a.net.out,
               VAL BYTE this.node, no.of.nodes)
-- bidirectional ring router process
-- "own packet" ring algorithm
-- with N size message buffer in Network8 process
-- ring must continue to circulate even with a blocked node
... include local procedures
CHAN OF BCUXR c.out, c.in, a.in, a.out :
PAR
  Sender(from.app, c.in, a.in, this.node, no.of.nodes)
  Collect(c.out, a.out, to.app)
  Network8(c.net.in, c.net.out, c.in, c.out, this.node, no.of.nodes)
  Network8(a.net.in, a.net.out, a.in, a.out, this.node, no.of.nodes)
:

```

The P-DB4GL ring router differs in some respects from the TRANSNET router described in [Welch89]. The TRANSNET router is used to provide deadlock-free networking for a channel multiplexing service already demonstrated to be deadlock-free. Free flow routing algorithms in cyclic topologies, such as rings, are prone to deadlock. The controlled flow "own packet" algorithm provided in [Welch89] is demonstrated to provide deadlock-free routing, assuming the channel multiplexing service it supports is also deadlock-free. The behaviour of the Transmit and Route processes in TRANSNET (corresponding to the Transmit and Receive processes in the P-DB4GL Network8 router), is defined so that the ring of packets would stop circulating if the application process at one of the nodes was unable to receive a message destined for it, hence blocking that node. Because of the "handshaking" protocol of the deadlock-free TRANSNET channel multiplexing, a node could not be blocked indefinitely, and the ring of packets would be allowed to continue circulating.

For the purposes of P-DB4GL network routing, no assumptions are made about the behaviour of applications connected to the router processes. In the event of an application process at some node becoming unable to receive a message from the router process, it was thought undesirable that such a blocked node should prevent the ring of packets from circulating. The node might be blocked for a number of reasons; an application process may have deadlocked, or else it might simply be suspended because of a large I/O delay. To provide continuous packet circulation, the Network8 router process was provided with a Buffer process, and small modifications were made to the Receive and Transmit processes. When the Receive process received a message destined for its node, it would first read a token on the Buffer's free

channel indicating the status (free or blocked) of the node. If blocked, the message would then be forwarded to Transmit and forced to circulate around the ring again. However, a side effect of this modification is the possibility that some messages may never be taken off the ring, but forced to continuously circulate. Therefore, the Buffer process was further modified, so that N messages (where N is the ring size) could be held in the Buffer before a node became blocked. When full, the N-size Buffer would flush out N messages to the waiting application process before accepting any more messages from the ring of packets.

```

PROC Transmit(CHAN OF BCUXR app.in, thru.in, net.out,
              VAL BYTE this.node, no.of.nodes)
  VAL INT nodes.less.one IS ((INT no.of.nodes) - 1) :
  BYTE empty.length, message.length :
  [max.BCUXR.length]BYTE message :
  [1]BYTE empty.message :
  BOOL running :
  SEQ
  -- transmit "own" empty message
  net.out ! zero::empty.message
  running := TRUE
  WHILE running
  SEQ
  -- copy through (n - 1) messages
  SEQ m = 0 FOR nodes.less.one
  SEQ
  thru.in ? message.length::message
  net.out ! message.length::message
  -- "own" packet has now returned
  thru.in ? message.length::message
  IF
  message.length = 0 (BYTE) -- "own" packet is free
  PRI ALT -- check if app message is waiting
  app.in ? message.length::message
  -- app message ready to go
  SKIP -- "own" packet used, app message is sent
  TRUE & SKIP -- app not ready to send
  SKIP -- "own" packet not needed, empty message sent
  TRUE -- "own" packet is still in use
  -- message not yet delivered
  SKIP
  -- send out (used or not used) "own" packet
  net.out ! message.length::message
  :
```

```

PROC Receive(CHAN OF BCUXR net.in, thru.out, app.out,
             CHAN OF BOOL free.in,
             VAL BYTE this.node )
-- if buffer not free to accept message,
-- then, send message round the ring
BYTE message.length :
[max.BCUXR.length]BYTE message :
[1]BYTE empty.message :
BOOL running :
BOOL ready :
SEQ
  ready := FALSE
  running := TRUE
  WHILE running
    PRI ALT
      -- accept buffer status
      free.in ? ready -- True or False
      SKIP
      -- accept message at this node
      net.in ? message.length::message
      VAL BYTE destination.node IS message[1] :
      IF
        message.length = zero
          -- empty message, forward through
          thru.out ! zero::empty.message
          NOT ready -- this node is blocked
          -- circulate message
          thru.out ! message.length::message
          -- test for this node
          (ready AND (destination.node = this.node))
          PAR
            -- pass message up to application
            SEQ
              app.out ! message.length::message
              free.in ? ready -- check new buffer status
              -- replace with empty message
              thru.out ! zero::empty.message
          -- else
          TRUE
            -- forward message through
            thru.out ! message.length::message
    :

```

To test the performance of the P-DB4GL routing code a number of test configuration were constructed: composed of rings of 16-bit T212 processors in sizes of two, four, and eight nodes. To test the performance of the Network component of the Router process, the test configuration Net16 was used (Figure J4). Net16 has a Network process on each node. Node 0 has a test harness process, Netest10, that constructs and sends BCUXR messages; these are all addressed to node 0, and Netest10 measures the time it takes each message to travel the full distance around the ring. Netest10 records the total number of its own messages sent and received during a test run, and calculates the minimum, maximum, and average message

communication times. Node 1 has a test harness process, Netest9, that maintains and displays a count of all BCUXR messages received at its node during a test run. At the remaining nodes in the ring, the test harness process Rtest4 generates and sends random varying size (0..256 bytes) BCUXR messages, and acts as a sink for all messages delivered to its node.

Tables I and II show the performance figures obtained from Net16 with heavily loaded rings. Table I shows the results from harness Netest10 when BCUXR messages of sizes 3, 51 and 256 are sent around an eight node ring loaded with messages by Rtest4 harnesses. It can be seen that the average time taken to deliver a BCUXR message a distance of eight nodes does not vary much with different message sizes. The 51 byte message size is typical of messages in P-DB4GL applications, and gives an average time of approximately 4.8 milliseconds. This is equivalent to 0.6 milliseconds per node and represents a unidirectional data transfer rate at each link of about 83 Kbyte/s. With 256 byte messages, an average time of approximately 6.4 milliseconds represents about 312 Kbyte/s at each link. In the worst possible case, with the ring fully saturated with the largest size messages, the average communication times would increase slightly, but the data transfer rate would increase too. Table II shows the number of messages received at harness Netest9 during one minute test runs. In the Table II test runs, the Netest10 process did not send any messages into the ring, and the figures show the capacity of the ring when loaded by two and six busy nodes (for four and eight node rings respectively).

Test configuration Net18 (Figure J5) was used to measure the performance of the BCUX Converter component of the Router process. Both nodes in this configuration have a Converter process that converts the BCUX messages to BCUXR protocol messages for transmission over the connecting 20 Mbit/s link. Node 0 has User Harness (see Appendix F) Dbhrns10 operating in interactive mode to supply BCUX messages. Node 1 has: an Echo process, that returns an identical BCUX message with the source and destination object identifiers reversed; and a BCUX Display process, that drives a terminal connected to the B006 board upon which the node 1 T212 processor is mounted. Table III shows typical conversion and communication times for different sizes of BCUX request-reply message pairs. Also shown are equivalent times for: communication of BCUX messages without conversion to the BCUXR protocol (ie no BCUX Converter process); and communication with neither conversion nor display of echoed messages (ie

no BCUX Display process either). Although Echo and Display occur concurrently, the improvement in communication time without display of echoed messages probably results from reduced processing load on Node 1 caused by the removal of the Display process.

Test configuration Net19 (Figure J6) was used to measure the performance of the full Router process, with both Network and BCUX Converter processes. Table IV shows typical communication times for different size BCUX request-reply message pairs sent a distance of one node. These times are for non-loaded rings with empty packets circulating at full speed. When used in P-DB4GL applications, it is likely that the Router ring would be heavily loaded with varying size messages, and the average communication times for request-reply message pairs would be slightly, though not much, worse than the times shown in Table IV.

In the course of testing the P-DB4GL Network8 router process, it was found that nodes often became blocked, and messages frequently had to be recirculated. The test harnesses used in the ring router testing could be used to both: simulate the behaviour of P-DB4GL applications; and load and saturate the ring. The maximum performance obtained from the ring router, in terms of messages carried per minute, was approximately similar to the number of messages communicated during test runs conducted on the prototype P-DB4GL test applications. No proof that the P-DB4GL routing is deadlock-free has been attempted. However, no further development work on this or similar message routing software is envisaged or recommended. Message routing on ring topologies with the current T-range of transputers introduces unacceptable delays for P-DB4GL applications. Alternative topologies using the T-range may be suitable, but the new generation of transputer products (H-range) provide support for message passing, therefore additional P-DB4GL routing software is not required.

Table I Ring Router Message Delivery Times

Test configuration Net16 - bidirectional (Network9 process) ring composed of eight T212 processors at 20 Mbit/s link speed.

Figures show the delivery time for a BCUXR message sent a distance of eight nodes.

Timings are in timer ticks (one tick = 64 microseconds).

Message size in Bytes (data size in parentheses)	3 (0)	51 (48)	256 (253)
Minimum delivery time	25	28	69
Maximum delivery time	151	183	186
Average delivery time	73.43	75.2	100.06
Total number of messages delivered per minute	11 563	11 029	8 432

Table II Ring Router Message Capacities

Test configuration Net16 - unidirectional (Network8 process) rings and bidirectional (Network9 process) rings composed of four and eight T212 processors, tested at link speeds of 10 and 20 Mbit/s.

Test runs of one minute duration, with random varying size (0..256 Bytes) messages used to load the ring router.

Figures show the total number of BCUXR messages received at test harness Netest9 during a one minute test run.

Ring size and Link Speed (Mbit/s)	4 nodes		8 nodes	
	10	20	10	20
Unidirectional ring router messages received per minute	19 232	23 967	14 952	19 192
Bidirectional ring router messages received per minute	24 811	28 017	20 591	22 611

Table III Performance of the BCUXR Converter

Test configuration Net18 - two T212-20 processors at 20 Mbit/s link speed.
 Typical times for conversion of a BCUX message to BCUXR protocol
 followed by transmission and echo of the request-reply message pair over a
 link.

Time in microseconds.

Message size :	BCU data only	0	48	145
(Bytes)	BCUX total size	(28)	(76)	(173)
	BCUXR total size	31	79	176
Message conversion and communication time		832	1 088	1 408
No BCUX conversion - communication time only		(384)	(576)	(832)
No conversion and no display of echo - communication time only		(256)	(448)	(704)

Table IV Ring Router Performance

Test configuration Net19 - four node and eight node T212 bidirectional rings
 with full Router (BCUXR Converter and Network9 processes) at 20 Mbit/s
 link speed.

Typical conversion and communication times for message distance of one
 node for request-reply message pair (total distance, two nodes).

Time in microseconds.

Ring size:	4 node			8 node			
Message size :	BCU data only	0	48	145	0	48	145
(Bytes)	BCUX total size	28	76	173	28	76	173
	BCUXR total size	31	79	176	31	79	176
Message conversion and communication time		3 392	4 032	4 864	3 648	4 032	5 248
Conversion and communication time - with no echo display		2 752	3 264	4 096	3 328	3 648	4 224

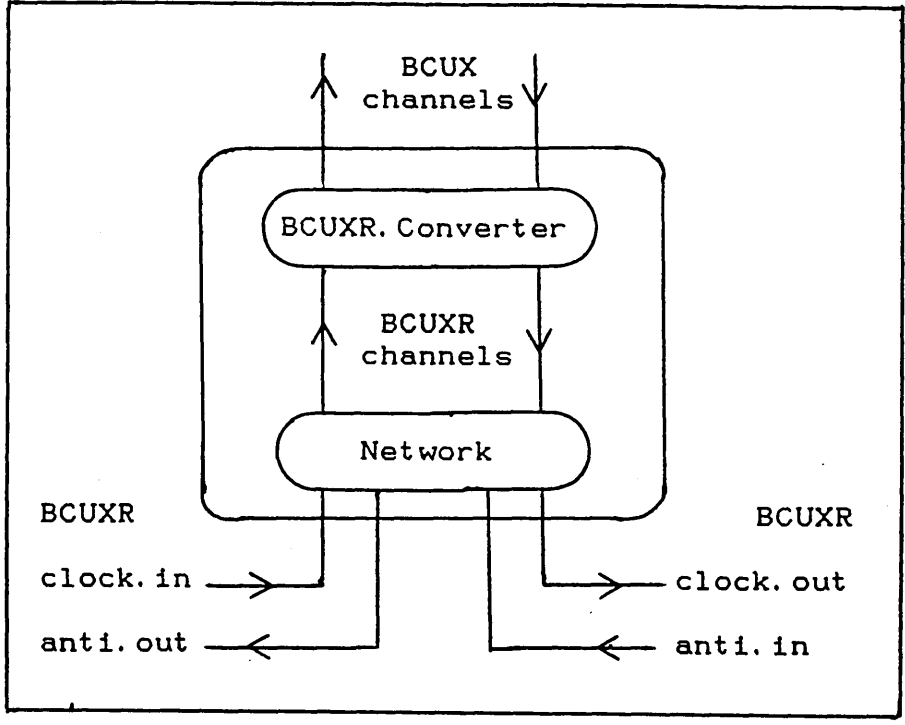


Figure J1 - P-DB4GL Router Process

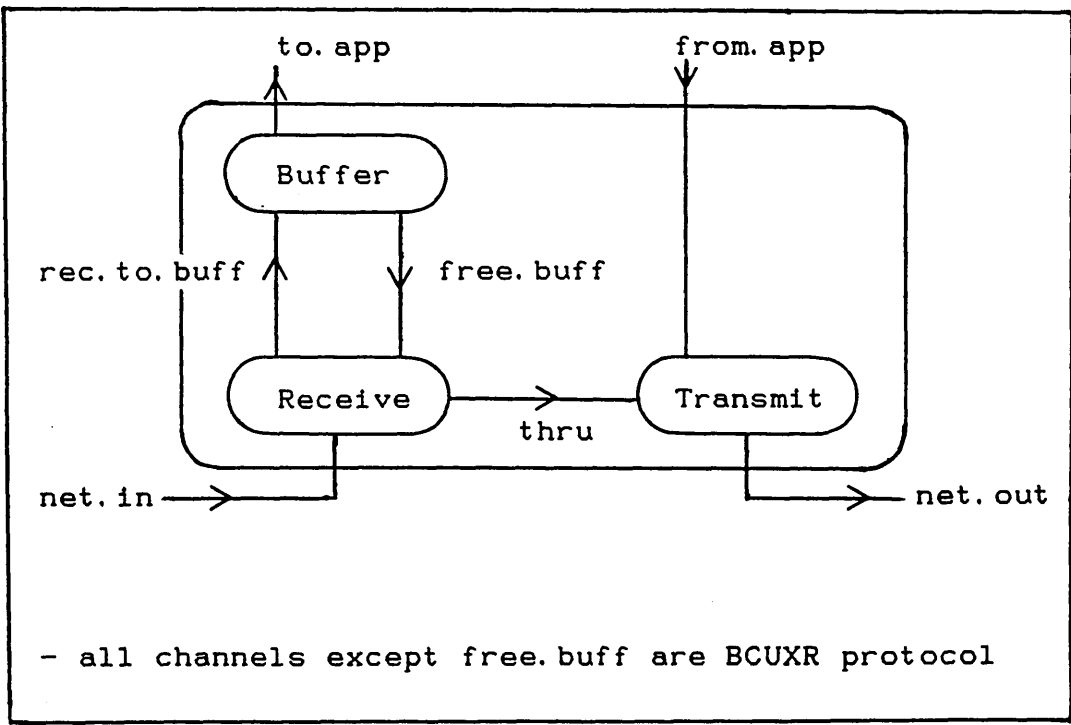


Figure J2 - Network8 Ring Router Process

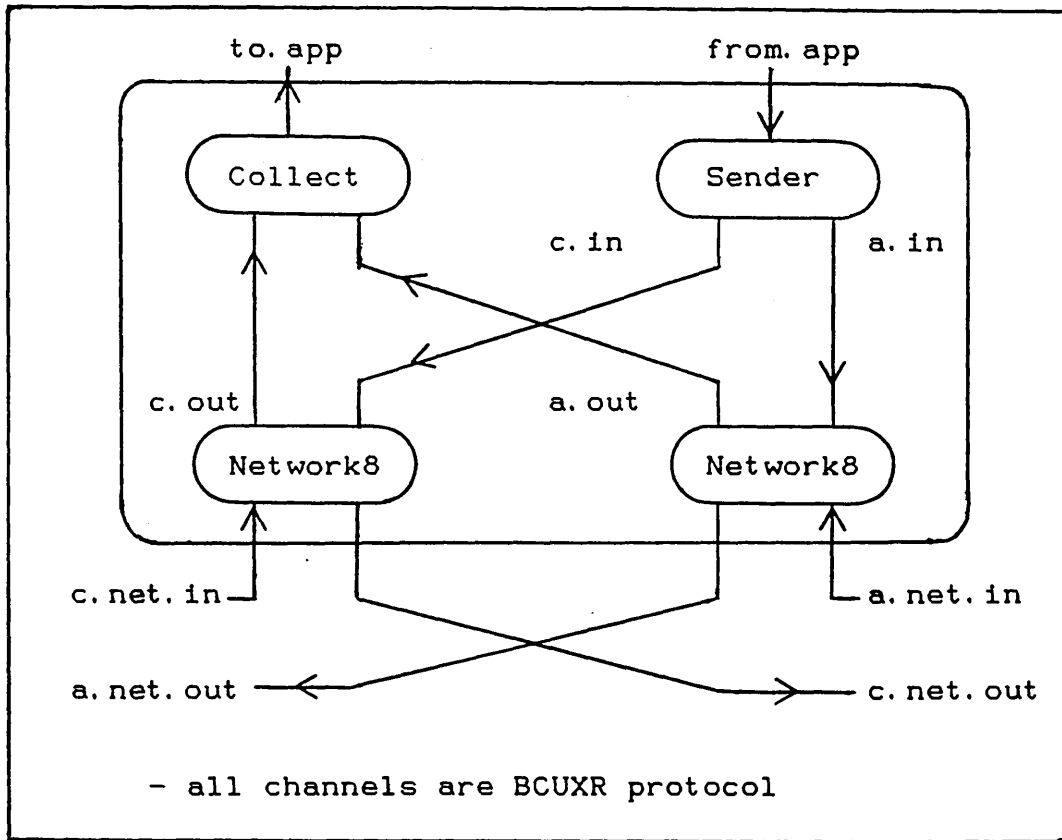


Figure J3 - Network9 Ring Router Process

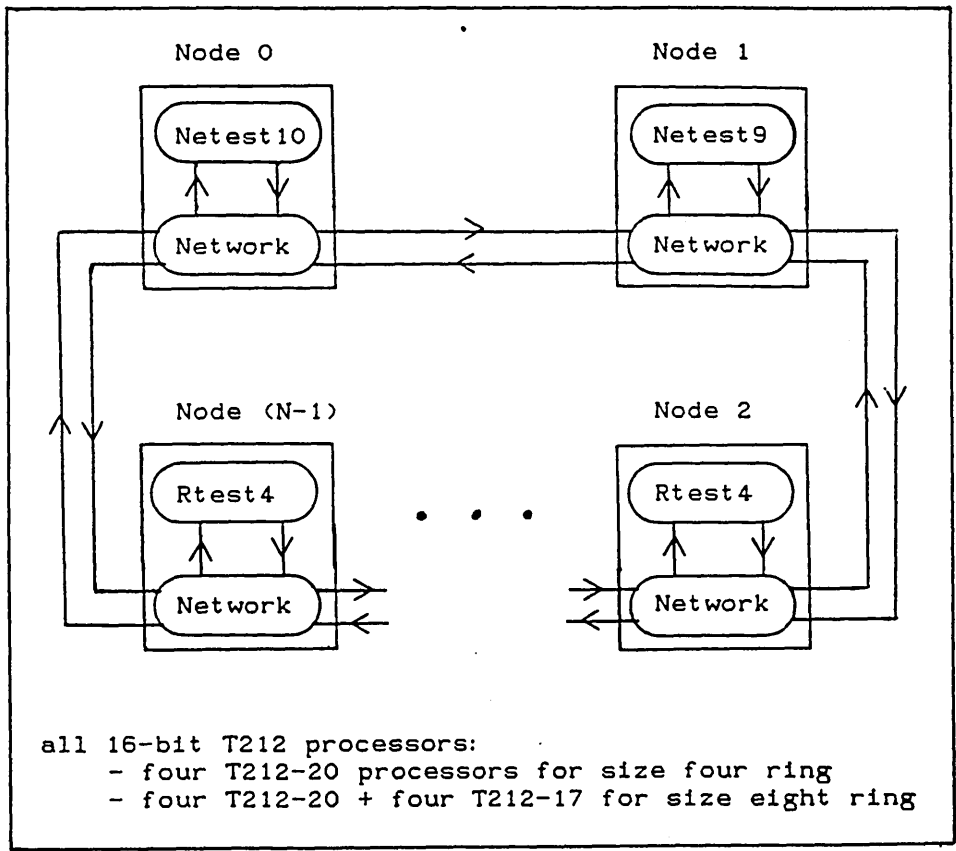


Figure J4 - Net16 Ring Routing Test Configuration

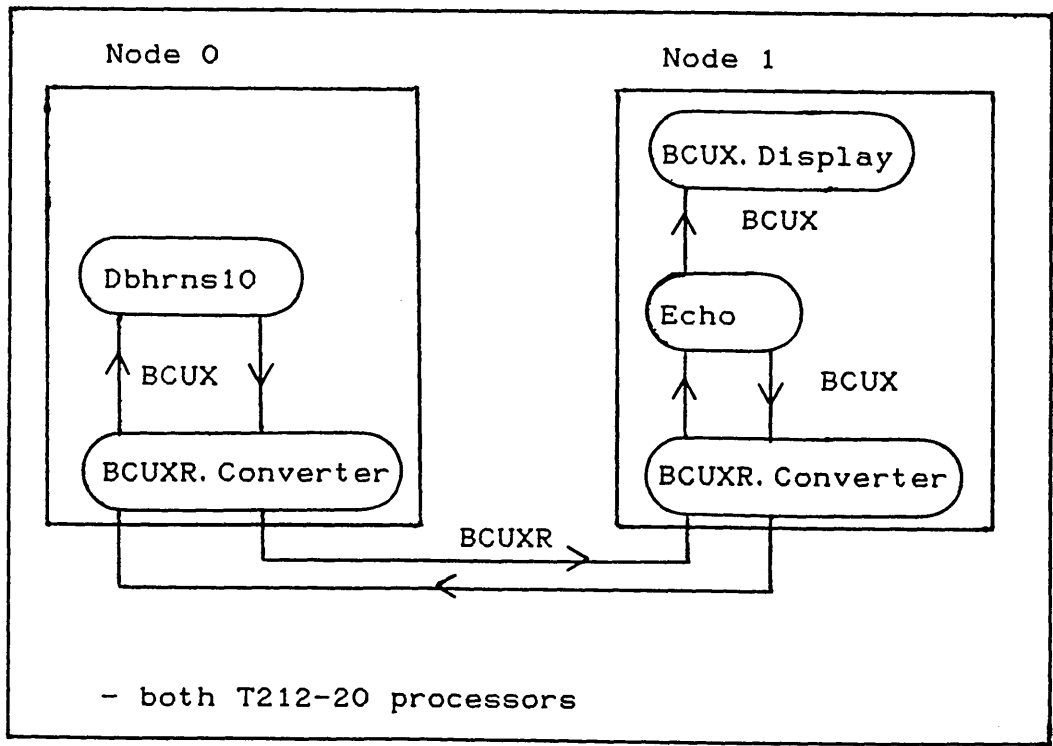


Figure J5 - Net18 Ring Routing Test Configuration

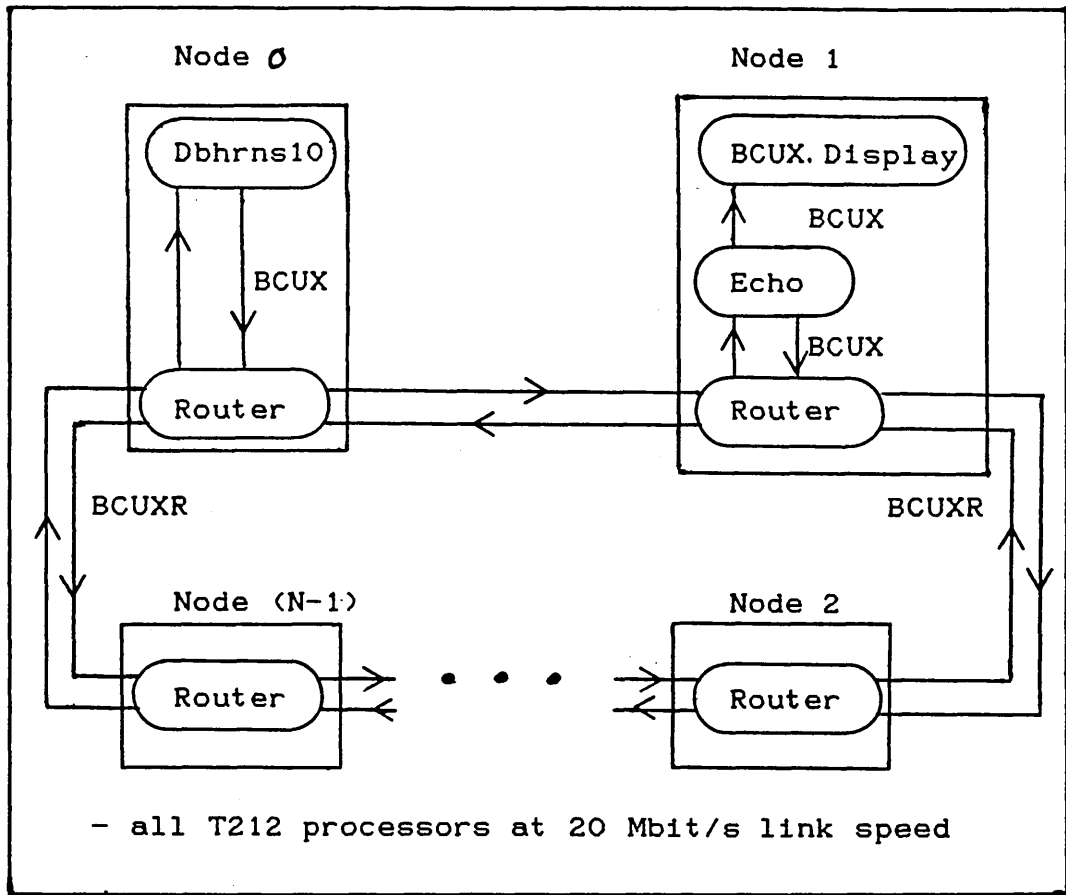


Figure J6 - Net19 Ring Routing Test Configuration

Appendix K

System Constants

SYSTCONS.INC

```
--{{{ LIB dbms.system.constants
--{{{ systemconstants
VAL INT object.id.length IS 8 :
VAL INT object.type.length IS 4 :
VAL INT object.number.length IS (object.id.length -
                                object.type.length) :
VAL INT entity.id.length IS object.number.length :
VAL INT attribute.id.length IS object.number.length :
VAL INT bcuXX.messege.limit IS 255 :
VAL INT bcuXX.node.id.length IS 1 :
VAL INT bcuXX.protocol.overhead IS (bcuXX.node.id.length * 2) :
VAL INT bcuX.protocol.overhead IS (object.id.length * 2) :
VAL INT bcu.message.details IS 4 : -- mode + status + io + val.size
VAL INT bcu.protocol.overhead IS ((entity.id.length +
                                attribute.id.length) +
                                bcu.message.details) :
VAL INT total.message.overhead IS ((bcuXX.protocol.overhead +
                                bcuX.protocol.overhead) +
                                bcu.protocol.overhead) :
VAL INT bcu.max.value IS (bcuXX.messege.limit -
                          total.message.overhead) :
VAL INT max.attribute.value.length IS 80 :
VAL INT max.no.of.fields IS 20 :
VAL INT max.no.key.fields IS 19 :
VAL INT max.record.length IS (max.no.of.fields *
                              max.attribute.value.length) :
VAL INT max.key.length IS 120 : -- Micro Focus COBOL limit
VAL INT max.no.schema.links IS 100 :
VAL INT max.no.entities IS 100 :
VAL BYTE disc.terminate IS 1 (BYTE) :
VAL BYTE disc.request IS 0 (BYTE) :
VAL BYTE disc.reply IS 0 (BYTE) :
VAL BYTE null.char IS 0 (BYTE) :
VAL BYTE numeric IS '9' (BYTE) :
VAL BYTE character IS 'X' (BYTE) :
--}}}}
--}}}}
```

CODEVALS.INC

```
--{{{ LIB dbms.system.codes
--{{{ dbms.system.codes
--{{{ system.codes
VAL BYTE entity.action IS 'E' :
VAL BYTE attribute.action IS 'A' :
VAL BYTE closed IS 'C' :
VAL BYTE opened IS 'O' :
VAL BYTE input.output.mode IS 'O' :
VAL BYTE input.mode IS 'I' :
VAL BYTE output.mode IS 'O' :
VAL BYTE not.available IS 'N' :
```

```

VAL BYTE true.stat IS 'T' :
VAL BYTE false.stat IS 'F' :
VAL BYTE locked.rec IS 'D' :
VAL BYTE locked.file IS 'A' :
VAL BYTE terminate IS 'X' :
--}}}
--{{{ object.id.codes
-- object identity codes
VAL []BYTE kernal IS "KRNL" :
VAL []BYTE prime IS "ENTY" :
VAL []BYTE couple IS "CPLE" :
VAL []BYTE schema IS "SCHM" :
VAL []BYTE filer.h IS "FLRH" :
VAL []BYTE user.h IS "USRH" :
--}}}
--{{{ entity.action.codes
VAL []BYTE null.action IS "0000" :
VAL []BYTE open.entity IS "0001" :
VAL []BYTE close.entity IS "0002" :
VAL []BYTE get.entity IS "0003" :
VAL []BYTE get.next.entity IS "0004" :
VAL []BYTE locate.entity IS "0005" :
VAL []BYTE store.entity IS "0006" :
VAL []BYTE delete.entity IS "0007" :
VAL []BYTE clear.entity IS "0008" :
VAL []BYTE clear.data IS "0009" :
VAL []BYTE save.entity IS "0010" :
VAL []BYTE restore.entity IS "0011" :
VAL []BYTE get.eq.entity IS "0012" :
VAL []BYTE get.gt.entity IS "0013" :
VAL []BYTE save.key.entity IS "0014" :
VAL []BYTE restore.key.entity IS "0015" :
VAL []BYTE clear.partial.entity IS "0016" :
VAL []BYTE push.key.entity IS "0017" :
VAL []BYTE pop.key.entity IS "0018" :
VAL []BYTE init.queue IS "0019" :
VAL []BYTE clear.attribute IS "0020" :
--}}}
--{{{ error.codes
VAL BYTE no.error IS 0 (BYTE) :
VAL BYTE err.get IS 50 (BYTE) :
VAL BYTE err.get.next IS 51 (BYTE) :
VAL BYTE err.locate IS 52 (BYTE) :
VAL BYTE err.store IS 53 (BYTE) :
VAL BYTE err.delete IS 54 (BYTE) :
VAL BYTE err.write IS 55 (BYTE) :
VAL BYTE err.rewrite IS 56 (BYTE) :
VAL BYTE err.entity.closed IS 57 (BYTE) :
VAL BYTE err.illegal.action IS 58 (BYTE) :
VAL BYTE err.open IS 59 (BYTE) :
VAL BYTE err.invalid.attribute IS 61 (BYTE) :
VAL BYTE err.locate.eq IS 62 (BYTE) :
VAL BYTE err.locate.gt IS 63 (BYTE) :
VAL BYTE err.dec IS 64 (BYTE) :
VAL BYTE err.filespec IS 65 (BYTE) :
VAL BYTE rec.lock IS 71 (BYTE) :
VAL BYTE file.lock IS 72 (BYTE) :
VAL BYTE err.entity.identifier IS 80 (BYTE) :
--{{{ schema values

```

```
VAL BYTE illegal.entity IS 91(BYTE) :
VAL BYTE illegal.attribute.op IS 92(BYTE) :
VAL BYTE illegal.entity.op IS 93(BYTE) :
VAL BYTE locked.mode IS 94(BYTE) :
VAL BYTE err.realise.fail IS 95(BYTE) :
--}}}
--{{{ message failures
VAL BYTE err.source.object.type IS 251 (BYTE) :
VAL BYTE err.object.identifier IS 252 (BYTE) :
VAL BYTE err.node.identifier IS 253 (BYTE) :
VAL BYTE err.message.fail IS 254 (BYTE) :
--}}}
VAL BYTE err.terminate.fail IS 255 (BYTE) :
--}}}
--}}}
--}}}
```

SYSTVALS.INC

```
#INCLUDE "c:\dbms\syscons.inc"
#INCLUDE "c:\dbms\codevals.inc"
```

NETVALS.INC

```
-- constant values for network code
VAL INT max.no.of.nodes IS 256 :
VAL INT max.objects.per.node IS 20 :
VAL INT max.BCUXR.length IS 256 :
```

Appendix L

Channel Protocols

DBPCOLS.INC

```
--{{{ LIB protocol.decs
--{{{
#include "c:\dbms\syscons.inc"
--{{{ tds channels
PROTOCOL SCREEN IS BYTE :
PROTOCOL KEYBOARD IS INT :
--}}}
--{{{ basic.comm.unit.protocol
PROTOCOL Eh.number IS [entity.id.length]BYTE :
PROTOCOL Eh.mode IS BYTE :
PROTOCOL Eh.operation IS [attribute.id.length]BYTE :
PROTOCOL Eh.stat IS BYTE :
PROTOCOL Eh.io IS BYTE :
PROTOCOL Eh.val IS BYTE::[]BYTE :
-- BCU is a sequential protocol composed of...
-- BCU IS Eh.number; Eh.mode; Eh.operation; Eh.stat; Eh.io; Eh.val :
PROTOCOL BCU IS [entity.id.length]BYTE;
                BYTE;
                [attribute.id.length]BYTE;
                BYTE; BYTE; BYTE::[]BYTE :
--}}}
--{{{ basic.comm.unit.eXtended.protocol
-- BCUX is a sequential protocol composed of...
-- BCUX IS Source.object; Destination.object;
--     Eh.number; Eh.mode;
--     Eh.operation; Eh.stat; Eh.io; Eh.val :
PROTOCOL BCUX IS [object.id.length]BYTE;
                [object.id.length]BYTE;
                [entity.id.length]BYTE;
                BYTE;
                [attribute.id.length]BYTE;
                BYTE; BYTE; BYTE::[]BYTE :
--}}}
--{{{ tuple.object.protocols
-- <field.specification> = <field.id> <field.value>
-- <field.id> = INT
-- <field.value> = INT::[]BYTE
-- <dump.string> = INT::[]BYTE
PROTOCOL TUPLE.REQ
    CASE
        T.get; INT -- <field.id>
        T.put; INT; INT::[]BYTE -- <field.specification>
        T.fill; INT::[]BYTE -- <dump.string>
        T.dump
        T.terminate
    :
PROTOCOL TUPLE.REPLY
    CASE
        T.return.field; INT::[]BYTE -- <field.value>
        T.put.ok
        T.fill.ok
        T.return.dump; INT::[]BYTE -- <dump.string>
```

```

T.terminate.ok
T.error.field.id
T.error.field.empty
T.error.value.underflow
T.error.value.overflow
T.error.dump; INT::[]BYTE      -- <dump.string>
T.error.terminate
:
--}}}
--{{{  filer.object.protocols
-- <file.description> = <i.o.status> <record.description>
-- <i.o.status> = BYTE
-- <record.description> = <record.length> <key.length>
-- <record.length> = INT
-- <key.length> = INT
-- <key> = INT::[]BYTE
-- <record> = INT::[]BYTE
PROTOCOL FILER.REQ
CASE
  F.open; BYTE; INT; INT      -- <file.description>
  F.close
  F.read; INT::[]BYTE        -- <key>
  F.read.next
  F.write; INT::[]BYTE       -- <record>
  F.rewrite; INT::[]BYTE     -- <record>
  F.delete; INT::[]BYTE      -- <key>
  F.start.equal; INT::[]BYTE -- <key>
  F.start.greater; INT::[]BYTE -- <key>
  F.start.not.less; INT::[]BYTE -- <key>
  F.terminate
:
PROTOCOL FILER.REPLY
CASE
  F.open.ok
  F.close.ok
  F.read.ok; INT::[]BYTE     -- <record>
  F.read.next.ok; INT::[]BYTE -- <record>
  F.write.ok
  F.rewrite.ok
  F.delete.ok
  F.start.ok
  F.terminate.ok
  F.at.end
  F.invalid.key
  F.error
:
--}}}
--}}}
--}}}

```

HRNSPCOLS.INC

```

--{{{  timing protocols
PROTOCOL TIME.CONTROL
CASE
  TC.reset      -- reset all timers, switch all transmissions off
  TC.bcu.on     -- toggle on bcu timing

```



```

TC.bcu.off      -- toggle off bcu timing
TC.start.trans  -- time start of transaction
TC.end.trans    -- time end of transaction, transmit trans time
TC.transmit.on  -- toggle on timings transmission
TC.transmit.off -- toggle off timings transmission
TC.current.time -- transmit current time
:
PROTOCOL TIME.RESULT
CASE
  TR.bcu; INT      -- <time>
  TR.trans; INT    -- <time>
  TR.current; INT  -- <time>
:
--}}}
--{{{ batch control protocol
-- BC.bcu.mess; <eh.number> <eh.mode> <eh.operation>
--           <eh.stat> <eh.io> <eh.val>
PROTOCOL BATCH.CONTROL
CASE
  BC.start.app    -- start of application
  BC.start.trans  -- start of transaction
  BC.end.trans    -- end of transaction
  BC.end.app      -- end of application
  BC.bcu.mess; [entity.id.length]BYTE; BYTE;
               [attribute.id.length]BYTE;
               BYTE; BYTE; BYTE::[]BYTE
:
--}}}

```

NETPCOLS.INC

```

-- network protocols
-- BCUXR is a counted array protocol for ring topology
-- first BYTE is the length of the following array containing message
-- [0]BYTE of the array is the source node
-- [1]BYTE of the array is the destination node
-- ring nodes are identified as 0..(N-1), where N is the
-- size of the ring
PROTOCOL BCUXR IS BYTE::[]BYTE :

```