



Kent Academic Repository

Fathy, Sherif Kassem (1991) *Exploring parallelism with object oriented database management system*. Doctor of Philosophy (PhD) thesis, University of Kent.

Downloaded from

<https://kar.kent.ac.uk/94340/> The University of Kent's Academic Repository KAR

The version of record is available from

<https://doi.org/10.22024/UniKent/01.02.94340>

This document version

UNSPECIFIED

DOI for this version

Licence for this version

CC BY-NC-ND (Attribution-NonCommercial-NoDerivatives)

Additional information

This thesis has been digitised by EThOS, the British Library digitisation service, for purposes of preservation and dissemination. It was uploaded to KAR on 25 April 2022 in order to hold its content and record within University of Kent systems. It is available Open Access using a Creative Commons Attribution, Non-commercial, No Derivatives (<https://creativecommons.org/licenses/by-nc-nd/4.0/>) licence so that the thesis and its author, can benefit from opportunities for increased readership and citation. This was done in line with University of Kent policies (<https://www.kent.ac.uk/is/strategy/docs/Kent%20Open%20Access%20policy.pdf>). If you ...

Versions of research works

Versions of Record

If this version is the version of record, it is the same as the published version available on the publisher's web site. Cite as the published version.

Author Accepted Manuscripts

If this document is identified as the Author Accepted Manuscript it is the version after peer review but before type setting, copy editing or publisher branding. Cite as Surname, Initial. (Year) 'Title of article'. To be published in *Title of Journal*, Volume and issue numbers [peer-reviewed accepted version]. Available at: DOI or URL (Accessed: date).

Enquiries

If you have questions about this document contact ResearchSupport@kent.ac.uk. Please include the URL of the record in KAR. If you believe that your, or a third party's rights have been compromised through this document please see our [Take Down policy](https://www.kent.ac.uk/guides/kar-the-kent-academic-repository#policies) (available from <https://www.kent.ac.uk/guides/kar-the-kent-academic-repository#policies>).

EXPLORING PARALLELISM WITH AN OBJECT ORIENTED
DATABASE MANAGEMENT SYSTEM

SHERIF KASSEM FATHY

A Thesis submitted for the Degree of
Doctor of Philosophy in Computer Science
University of Kent at Canterbury
England

March 1991

Dx 94313

F138918



Abstract

The object oriented approach to database management systems aims to remove the limitations of the current systems by providing enhanced semantic capabilities and more flexible facilities, including the encapsulation of operations as well as data in the specification of an object. Such systems are certainly more complex than existing database management systems. Although, they are complex, the current object oriented database management systems are built for Von-Neumann (purely sequential) machines. Such implementation inevitably leads to major problems involving efficiency and performance. So, new techniques for implementation need to be investigated.

One possible solution for the efficiency, and performance problems is to use parallel processing techniques. Thus, the aim of this research is to propose aspects in which parallel processing can be introduced within the scope of object oriented database management systems and identify ways in which the performance can be improved. A prototype of the main components of an object oriented database system called KBZ has been implemented to test out some of the parallel processing aspects.

The thesis starts with an introduction and background to the research. It then describes major parallel system architectures for an object oriented database management system. Techniques such as distributing a large volume of data among various processors (transputers), performing processing in the background of the system to reduce response time, and performing input/output parallel processing are presented. The initial prototype, PKBZ version-1, is then described; in particular, the logical and physical representation of object classes, how they communicate through message sending, and the different types of message supported. Two prototype versions exist. The initial prototype was designed to investigate the parallel implementation and general functionality of the system. The second version provides greater flexibility and incorporates enhanced functionality to allow experimentation. The enhancements in the second version are also discussed in the thesis, and the experimental results using different transputer configurations are illustrated and analyzed.

Acknowledgements

At the completion of this work, I would like to express my deep gratitude to Dr. Elizabeth Oxborrow. I feel most in debit to her constructive criticism, valuable guidance, and patience during the research.

I would like to thank Mr. R. E. Jones for great help during the implementation phase.

Thanks are due to all the staff of the Computing Laboratory, and especial thanks to the people in Meiko Room at the University of Kent, exclusively Prof. Peter Welch, Tony Curtis, Vedat Demiralp, and Godfrey Paul.

I am extremely grateful to my wife and to my children Samer and Nader, for their support, encouragement, and patience.

I would like to express my extremely sincere gratitude to my parents for their prayers, which have been the greatest benefit to me. Especial thanks to my brother engineer RabeH for his looking after my complex affairs during my research.

I would like to acknowledge the financial support provided by the Egyptian government throughout my research.

Especial thank to all my colleagues in my college at Egypt.

TABLE OF CONTENTS

CHAPTER 1: Research Background :

1.1. Introduction	1
1.2. Introduction to Object Oriented Database Management Systems	2
A. Data Modelling Constructs	2
B. Semantic Aspects	2
C. Data Integrity	3
D. Rules Storage Capabilities	3
1.2.1. New Non-traditional DBMSs	3
A. Adapted Relational Databases	4
B. Adapted Logic-based Databases	4
C. Object Oriented Databases	4
i. Non Object Oriented Languages	5
ii. Extended Object Oriented languages	5
1.3. The Potential of Object Oriented Approach	5
A. Object Concept and Encapsulation	6
B. Classes and Inheritance	8
C. Overloading, Overriding and Late Binding	9
1.4. THE KBZ Object Oriented Database System	10
1.4.1. Basic Objects	12
1.4.2. Structural Objects	12
1.4.3. Derived (Complex) Objects	17
A. Entity Aggregate	17
B. Subtypes	17
1.5. Introduction to Parallel Processing	19
1.5.1. Classification of Computer Systems	20
A. Single Instruction Single Data (SISD)	20

B. Single Instruction Multiple Data (SIMD)	20
C. Multiple Instruction Single Data (MISD)	20
D. Multiple Instruction Multiple Data (MIMD)	21
1.6. Hardware Architecture of The Transputer	21
1.7. Occam and The Transputer	23
1.8. Research Objectives	25
1.9. Thesis Layout	26

CHAPTER 2 : The State of The Art In Related Research :

2.1. Introduction	28
2.2. Object Oriented Database Management System	28
2.2.1. EXODUS DBMS	28
A. An Overall EXODUS System Architecture	28
B. EXODUS Conclusion	31
2.2.2. Iris DBMS	31
A. Iris System Structure	31
B. Iris Conclusion	33
2.3. Object Oriented Systems with Parallel Features	33
2.3.1. PRESTO	33
A. Why C++ with PRESTO	34
B. PRESTO object model	34
C. PRESTO Thread Class	35
D. PRESTO System Architecture	35
E. PRESTO Conclusions	36
2.3.2. Rekursiv : An Object-Oriented CPU	38
A. Rekursiv Architecture	38
B. Rekursiv Operation	39
C. Rekursiv Microcode	42
D. Rekursiv Applications	42

E. Rekursiv Conclusion	43
2.3.3. CHANCER	43
A. Chancer Message	43
B. Chancer Object	44
C. System Operation	46
D. Chancer Conclusion	46
2.4. Parallel Query Operation	47
A. General operation	47
B. Conclusion of Parallel Query Operation	49
2.5. General Conclusion	49

CHAPTER 3 : Parallel Processing Aspects In An Object Oriented

Database Management System :

3.1. Introduction	51
3.2. General Description of the PKBZ Architecture	51
3.2.1. External Management System (EMS)	51
3.2.2. Object Management System (OMS)	53
3.2.3. Storage Management System (SMS)	54
3.3. Different Aspects of Parallelism in PKBZ	54
3.3.1. Parallel on Transputers	55
A. Homogenous Parallel System Architecture	56
* Simple Homogenous Parallel System Architecture	56
* Complex Homogenous Parallel System Architecture	59
- Static Group	59
- Dynamic Group	61
B. Hybrid Parallel System Architecture	61
3.3.2. Parallel on a Single Transputer	62
3.3.3. Parallel on The Instructions	66
3.3.4. Parallel Data Distribution	67

3.3.5. Parallel on Background	69
A. Complete/Incomplete Parallel Processing	70
B. Object Update Parallel Processing	71
C. File Simulation Processing	73
3.4. Memory Object Grouping in PKBZ	78
3.4.1. Type of Memory Object grouping	78
A. Inheritance Structure Grouping	78
B. Non-Inheritance Structure Grouping	79
C. Random Grouping	80
D. Sequential Grouping	80
3.4.2. Allocation of Memory Object	81
A. Demanded Object	81
B. Inherit Structure Object	81
3.4.3. Deallocation of Memory Object	82
3.5. Conclusion	82

CHAPTER 4: PKBZ Implementation : Object Class Representation

4.1. Introduction	85
4.2. Logical Representation of the Object Class	86
4.2.1. Memory Object Representation	88
A. Object Identification	88
B. Property Names and Types of The Object Class	89
B.1. Basic Object Property Names and Types	89
B.2. Structural Object Property Names and Types ..	89
B.3. Entity Aggregate Object Property Names and Types	90
C. Inheritance Information	91
D. Constraints and Functional Properties	93
4.2.2. Memory Object Instances Representation	93

A. Basic Memory Object Instances Representation	94
B. Structural Memory Object Instances Representation	94
C. Derived Memory Object Instances Representation	96
D. Page Object Instances Representation	98
E. Instance Description	100
4.2.3. Object Methods and Constraints Representation	102
4.3. System Object Classes	103
A. OBJECT-ID Object Class	103
B. OBJECT-NAME Object Class	103
C. OSCHEMA Object Class	105
D. HAS-ONAME Object Class	105
E. HAS-OSCHEMA Object Class	105
4.4. Physical Representation of the Object Class	106
4.4.1. Instances Memory Object Transformation	107
4.4.2. Schema Memory Object Transformation	110
4.5. Database File Structure	111
4.5.1. Schema System Group	112
4.5.2. Instances System Group	113
4.5.3. Database System Group	114
4.6. Conclusion	116

CHAPTER 5: PKBZ Implementation : Messages

5.1. Introduction	118
5.2. The PKBZ Version-1 General Block Diagram	118
5.2.1. Hardware Channel Functions	120
5.3. PKBZ Object Oriented Database Operation	122
5.4. PKBZ Object Oriented Database Messages	125
5.4.1. System Messages	125
A. Create Database Message	127

B. Open Database Message	130
* File Simulation Process	133
* Analysis Order Process	135
C. Close Database Message	137
D. Create Object Class	138
5.4.2. Internal Messages	141
A. Open Object Message	141
B. Generate Surrogate Message	142
C. New Page Number Message	142
D. Update Object Message	142
E. Update Inherited By Message	143
5.4.3. Memory Object Message	143
A. Display Schema Description Message	143
B. Inherits Messages	144
C. Transaction Messages	144
C.1. Addition Messages	144
C.2. Retrieval Messages	145
* Check Instance Existence	145
* Get All Object Instances	146
* Get Specific Values	146
C.3. Updating Messages	146
C.4. Deletion Messages	147
5.4.4. Page Object Messages	147
A. Get Part Of Instance	148
B. Add Part Of Instance	148
5.5. Conclusion	148

Chapter 6: PKBZ Version-2 and Experimental Results

6.1. Introduction	150
-------------------------	-----

6.2. An Object Class Logical Representation (Memory Object)	
Version-2	151
6.3. Communication Channel Modifications	152
6.3.1. PKBZ version-2 General Block Diagram	152
6.3.2. Memory Object Communication Channels	154
6.4. Memory Object Instances	154
6.5. External Management System Batch Processing	155
6.6. PKBZ System Performance Evaluation	158
6.6.1. PKBZ Speed Up	161
6.7. Conclusion	162

Chapter 7: Conclusion and Future Researches

7.1. Introduction	165
7.2. Problem Areas With an OODBMS	165
7.2.1. Standard Definitions	165
7.2.2. Performance Problem	166
7.2.3. Secondary Storage Problems	166
7.2.4. Schema Evolution	166
7.3. Problem Areas With Parallel Processing	167
7.3.1. Deadlock Problem	167
7.3.2. Non-deterministic Behaviour	167
7.3.3. Static Language	168
7.3.4. Pattern Problem	168
7.3.5. Limited Number of Communication Channels	169
7.3.6. Imbalance Problem	170
7.3.7. Investment Limitation	170
7.3.8. Natural Human Limitation	170
7.4. Conclusion	171
7.5. Future Researches	174

Reference	176
-----------------	-----

APPENDIX A: PKBZ Object Oriented Database Management System

Constants and Variables Description

A.1. Miscellaneous Variables Description and Values	183
A.2. Object Type Identifier Values	184
A.3. Instance Type Identifier Values	184
A.4. Physical Instance Size According to Object Type Instance	185

APPENDIX B : PKBZ Schema Data Structure

B.1. The Object Identification Data Structure	186
B.2. The Object Class Property name and Types Data Structure	187
B.3. The Object Class Inheritance Information Data Structure	188
B.4. The Object Class Constraints Data Structure	189
B.5. The Object Class Functional Properties Data Structure	190
B.6. The Instances Description of Object Class Data Structure	191

APPENDIX C: PKBZ Instance Data Structure

C.1. The Data Structure of The Entity Set Instances and Attribute Set Instances of Integer Type	192
C.2. The Data Structure of Attribute Set Instances of String Type	192
C.3. The Data Structure of The Structural Object Instances	193
C.4. The Data Structure of The Entity Aggregate Object	

Instances	195
C.5. The Data Structure of The Page Object Instances	195
C.6. The Data Structure of The Page Data Region in Page Object Instance	196

APPENDIX D: PKBZ Protocols Descriptions

D.1. Protocol REQUEST.ORDER Definition	197
D.2. Protocol INSTANCE Definition	199
D.3. Protocol OBJECT.SCHEMA Definition	200

APPENDIX E: PKBZ Message Function and The Data Structure Descriptions

E.1. Common Variables Data Structure	201
E.2. REQUEST.ORDER Messages Data Structure	204
E.3. INSTANCE Messages Data Structure	207
E.4. OBJECT.SCHEMA Messages Data Structure	208

APPENDIX F: Database System Group Data Structure

209

APPENDIX G: External Management System Screens In PKBZ

G.1. The Basic Screen	210
G.2. The Message Screen	211
G.3. The Creation Screen	212
G.4. Create Structural Object Screen	213

CHAPTER 1

Research Background

1.1. Introduction :

Much recent research effort has been put into designing and building Object-Oriented DataBase Management Systems (OODBMSs) which accommodate a wide range of potential applications by supporting rich semantics and a high degree of flexibility and extensibility. Such researches include the Iris system [Fishman 87][Fishman 89] at Hewlett-Packard laboratories, ZEITGEIST [Ford 88] at Texas Instruments, ENCORE [Zdonik 86], EXODUS [Carey 86.a][Carey 87] at the University of Wisconsin, and other researches. OODBMS is a complex system. Although, it is complex, the current OODBMSs are built for Von-Neumann (purely sequential) machines. Such implementations inevitably lead to large problems involving efficiency and cost performance.

On the other hand, parallel processing technology provides huge processing capabilities that can be used in implementing an OODBMS. So, this research is being carried out to develop a prototype KBZ OODBMS [Oxborrow 88.a] using parallel processing technique. The developed prototype Parallel KBZ (PKBZ) is designed and implemented using Occam [Jones 86][Carling 88] as the parallel programming language on a Meiko computing surface¹.

In this chapter, the research background will be introduced. An introduction to object oriented database management systems (OODBMSs) will be described in subsection 1.2. The potential of the object oriented approach will be illustrated in subsection 1.3. The KBZ OODBMS model will be described in subsection 1.4. Then, an introduction to parallel processing will be discussed in subsections 1.5. The hardware architecture of the transputer will be illustrated in subsection 1.6. The relation between Occam and transputer will be described in subsection 1.7. The research objectives will be

¹ Meiko and computing surface are trademarks of Meiko Scientific Computing, and Occam is a trademark of the Inmos group of companies.

illustrated in subsection 1.8. Finally, the thesis layout will be described in subsection 1.9.

1.2. Introduction to Object Oriented Database Management Systems:

The requirements of most of the business applications can be satisfied quite well by the current DataBase Management System (DBMSs). However, some scientific and engineering applications, like Expert System, Computer Aided Design, Image Processing, and Pattern recognition, demand extra requirements from the database systems. These requirements are beyond the scope and the capabilities of the current DBMSs, even Relational Systems, since these applications are quite different in their modelling, data storage and retrieval. The features and the limitations of the current DBMSs can be summarized in the following :

A. Data Modelling Constructs :

The native data types available on most current DBMSs are integer, floating point, character, boolean, pointer ... etc. These data types may be sufficient for some applications in commercial systems such as Banking. These native data types are aggregated into tuples (in Relational DBMSs) [Date 83] or records (in the other DBMSs) [Codasyl 81.a.][Codasyl 81.b.][IMS]. The simple record structure underlying existing DBMSs is too rigid for engineering applications, since a low level representation in the engineering domain is unacceptable. It is essential to be able to work in terms of much more complex structured data types, with the ability to treat complete data structures, from simple matrices to complex layout diagrams, as a single entity.

B. Semantic Aspects :

In current DBMSs, the semantics of raw data are split between the database description and the application programs. Hence, some semantics not only are hidden in application programs, but also duplicated in different application programs to enforce semantic aspects. This aspect is due to the fact that the current DBMSs do not support the real system environments. Hence, it would be preferable to store the

semantics along with the data, thus making the data responsible for enforcing its correct use.

C. Data Integrity :

The current DBMSs store database schema separate from the actual data. Such separation is unnatural, since the data is meaningless without schema description. For example, in the engineering design, some sort of version control is needed. That is tracking changes of the data and schema with time, would be useful. This version control is important in coordinating a multi stage design process as well as return to a previous version. Such integration of data and schema is not provided in the current DBMSs.

D. Rules Storage Capabilities :

Most current DBMSs can handle and enforce some validation checks or constraints (i.e. values, ranges ... etc.). Such validation checks are stored in the schema. Artificial intelligence applications, and expert systems, for example, need to store not only data and validation checks, but also rules and constraints applied to the data. In real applications the number of rules and constraints applied to the data cannot be stored in the main memory, so a DBMS is needed. However, current DBMSs do not support general rules storage.

1.2.1. New Non-traditional DBMSs:

A number of database research efforts are being directed to build new database systems to accommodate a wide range of potential applications which are not supported by the current traditional database systems. Although, the goals of these research efforts are similar, and each uses some of the same mechanisms to provide extensibility, the overall approach of each research is quite different. In general, the new systems being designed and developed may be grouped into three main approaches :

- a. Adapted Relational Database Systems
- b. Adapted Logic-based Database Systems

A. Adapted Relational Database Systems :

This approach can be considered as object oriented front ends onto an underlying relational database system. That is, the relational database system is extended to provide new functionality. For example, STARBURST [Schwarz 86][Lindsay 87], and POSTGRES [Stonebraker 86][Rowe 87] are complete database systems, each with a different well-defined data model and query language. Each system aims to provide the capability for users to add extensions such as new abstract data types and access methods within the framework provided by their data model.

B. Adapted Logic-based Database Systems :

Some of the recent research has combined logic programming technology with database technology to produce Expert or Logic Database Systems [Kerschberg 86]. Some research effort has been based on Prolog and databases. For example, [Zaniolo 86.a] and [Gray 85][Gray 88] have investigated the combination of Prolog with Codasyl network.

It has to be mentioned that the adapted logic-based approach is aimed at solving the problem of large volumes of rules needed in Expert System environments [King 84], while the other two approaches are mainly concerned with the need to support semantically rich and complex data models.

C. Object Oriented Database Systems :

A lot of research is being carried out to implement Object Oriented Database Management Systems. Some of these researches are just prototypes : Iris [Fishman 89][Fishman 87], Gemstone [Purdy 87], ZEITGEIST [Ford 88], O2 [Bancilhon 88], POSTGRES [Stonebraker 86], ORION [Kim 89][Banerjee 87], ENCORE [Zdonik 85][Zdonik 86][Hornick 87], DAMOKLES [Dittrich 89], EXODUS [Carey 85][Carey 86.b][Carey 87], OZ+ [Weiser 89], and KBZ [Oxborrow 88.a]. But some researches have become commercial products : G-base [G-base 88], VISION [Caruso 87], and Ontos [Andrews].

There are variations in these systems with regard to the basic object types

which are supported and the mechanisms provided for deriving objects based on other objects, since as mentioned in [Atkinson 89] no standard terminology exists at present and the overall approach of each research is quite different. Furthermore, each system has different data model definition and each provides the end user with different accessing methods. For example, OZ+ [Weiser 89] is designed to support an object oriented office information system programming environment, while ORJON [Kim 89] provides various advanced functions required by applications from the CAD/CAM, and artificial intelligence domains. The variations in some different OODBMSs are illustrated in [Oxborrow 88.b]. A survey of OODBMSs may be found in [Thearle 89].

The object methods (implementation of an object's behaviour) may be written in a suitably extended standard programming language. The access to instance variables of the object is usually performed via a method. These methods, of course, can be executed in one of the several different languages.

In general, there are two approaches to designing and implementing the object oriented database management systems :

i. Non Object Oriented Languages :

These object oriented systems are being designed or prototyped in high level languages which are **not** object oriented languages. For example, OZ+ [Weiser 89] is being prototyped on a set of Sun-3/50 clients supported by a Sun-3/280 file server [Sun 86] and it is written in C [Kernighan 88].

ii. Extended Object Oriented languages :

These object oriented systems are usually designed using object oriented languages. For example, both EXODUS [Carey 87] and Ontos [Andrews] are based on C++ [Stroustrup 86][Dewhurst 89] object oriented language. While Gemstone [Purdy 87] is built in Smalltalk [Goldberg 84] object oriented language.

1.3. The Potential of the Object Oriented Approach :

The object oriented paradigm represents one of the most successful paradigms in

many areas of computer science. Object oriented paradigms have been popularized mainly through Smalltalk [Goldberg 83], building upon the concept of an "object class" introduced in Simula [Birtwistle 73]. This is not surprising, since it is quite natural to model real life entities as software objects.

The use of an object oriented paradigm in database management systems is aimed towards getting rid of the previous problems of the current database systems. Thus, it is better to discuss the main advantages of the object oriented database approach in the context of the discussion in subsection 1.2, so the main features of OODBMS are summarized in the following :

A. Object Concept and Encapsulation :

In the OODBMS, everything is viewed in terms of objects. An object can represent anything from single number, e.g. 10, to a complex entity such as EMPLOYEE or VEHICLE. Objects are entities that combine the properties of procedures and data, since they perform computations and save local state.

Object schemata and object data are logically integrated together inside each object. Such integration provides a high degree of "Data Integrity" which is not supported by the current DBMS as described in subsection 1.2.

Objects communicate and perform all computation via "messages". The behaviour of an object is captured in the messages to which an object responds. All of the action in the object oriented approach comes from passing messages between objects. Message sending is a form of indirect procedure call.

Messages, with any arguments that may be passed with the messages, constitute the public interface of an object. An object reacts to a message by executing the corresponding method. The principle of message sending is that calling programs should not make assumptions about the implementation and internal representation of data types that they use, so the underlying implementation can be changed without changing the calling programs. That is, every object comes endowed with a set of operators which are used to operate upon and change the state of the object. An object consists of an

interface part, which is public, and an implementation part, which is kept private.

Thus, this new database object concept overcomes the limitation described under "Data Modelling Constructs" in subsection 1.2., since each object is viewed as a complete entity in itself. Furthermore, by hiding the internal representation of the data, an object can provide the complex data structures to be represented, with the ability to treat complex data structures as a single entity in the sophisticated applications. This facility gets rid of the rigidity of the simple record structure underlying existing current DBMSs.

In addition, encapsulation is one of the major advantages in object oriented database systems, since it packages data together with related methods used to access or modify an object. Encapsulation has traditionally been important in computer science, in general, for the simple reason that it is necessary to decompose large systems into smaller encapsulated subsystems that can be more easily developed, maintained, and ported. Moreover, encapsulation provides a way that the system can store the semantic aspects along with data, thus making the data responsible for enforcing its correct use. That is, the semantics of raw data are not split between the database description and the application programs. Hence, the semantics are neither hidden in application programs nor duplicated in different application programs. Thus, the encapsulation concept gets rid of the limitation of "Semantic Aspects", described in subsection 1.2., of the current DBMSs.

The only way that an object is accessed by the outside world is by invoking one of its methods to access the data encapsulated in the object. Hence, the internal data structures and data inside an object are totally isolated. This provides system design modularity; it ensures that the data structures are easily accessible to the methods that use them, and provides a high degree of integrity for the data.

In OODBMS, rules can be stored in the form of methods. Such a facility provides a tool to overcome the limitation aspect of "Rules Storage Capabilities", described in subsection 1.2., of the current DBMSs. Moreover, methods can also be used to handle constraints on the data values.

B. Classes and Inheritance :

Classes describe collections of objects, so similar objects are grouped together into a class. Objects are organized into classes that contain the methods that the objects use to respond to a message.

All objects belonging to the same class are described by the same instance variables and the same methods. They all respond to the same messages. Thus, the class defines the data type, and the operations are methods the class responds to. That is, a class describes the form (instance variables) of its instances and the operations (methods) applicable to its instances. In general, a class has only one type, but a type may be associated with more than one class. A class contains the code and the data structure needed by its instances. This allows the class to be responsible for creating new objects and enables dynamic extensions to a running application program.

Inheritance is a relationship in object oriented systems which provides a way of sharing behaviour between objects. Every inheritance relationship has parents and children and properties which are inherited. The parents and children in an inheritance relationship could be classes or instances and the properties could be methods, instance variables, rules, constraints, values etc.

Classes are organized in a hierarchy or lattice, so that, they can inherit the structures and methods of their superclasses. Moreover, the type of an object not only may be based on other object types, but also may include new additional properties. Hence, objects with complex structure can be constructed. That is, classes can be organized into an implementational hierarchy so that objects in a lower class can inherit not only the internal data structure, but also methods of its parent class. The hierarchy can be either a tree structure, or multiple inheritance structure where a class can be a subclass of two independent classes [Schaffert 86]. It has to be mentioned that not all OODBMSs support multiple inheritance structure. Inheritance can be strict, that is a subclass can only add new methods to its parent's methods, or non-strict, when a subclass can override the inheritance mechanism and redefine some or all of its

parents' methods. That is the main advantage of the inheritance, since it provides a powerful code-sharing mechanism that can greatly reduce the effort required to create new object classes instead of new code having to be written.

C. Overloading, Overriding and Late Binding:

Consider the case when an application program wishes to apply the same methods to different objects with various types and the method is completely dependent upon the object type. The application program must know in advance the object type and tailor especial code according to this type. This implies that when a new complex object is specified, a new code must be tailored to adapt the current situation. In OODBMSs, this problem is solved by redefining the implementation of the operation for each of the types according to the type. This results in a single method name denoting different methods according to object type. That is, the system picks the appropriate implementation during the run-time according to object type. This process is called "overloading".

When a subclass can override the inheritance mechanism and redefine some or all of its parents methods, this process is called "overriding".

The main advantages of these properties are that the application programmer does not have to worry about the different system codes necessary to implement the method. In addition, the programmer's code is simpler as there is no type checking to perform the relevant system code. That is, there is no 'case' statement checking on types. Thus, the programmer's code is more maintainable, since the same programmer's code will continue to run without modification even when the same object changes its type.

In order to provide this new functionality, the type implementors still write the same system code, but it is not the application program's responsibility. So, the system cannot bind operation names to the method before the run time operation. Therefore, operation names have to be resolved at run-time. This delayed mechanism is called "late binding".

1.4. THE KBZ Object Oriented Database System :

As mentioned before, this research is being carried out to develop a prototype parallel KBZ (PKBZ) OODBMS, so in this subsection an overview of the KBZ OODBMS together with the examples will be discussed.

KBZ is an OODBMS which has been introduced by [Oxborrow 88.a]. KBZ has been influenced by system specification language Z which was introduced by Abrial and developed at Oxford University [Hayes 87]. Since KBZ is the kernel of this research, the main features of the KBZ design will be illustrated in this subsection. The initial version of KBZ was designed for a traditional database environment, in which it is common for large numbers of instances belonging to the same class to exist. So, PKBZ is also designed for this. The "AirLine Transportation", Fig.1.1. will be used as an illustrative example.

Each object type is represented by a schema which defines the properties (both data and functional) of objects of that type. An object schema is applied to both instances of the object and instances of objects which inherit the schema.

The naming conventions in KBZ objects are summarized in the following ² :

- i . A schema name is written in uppercase. For example, CREW, PLAN are schema names
- ii . The set of instances of the object represented by a schema is referred to by using the same schema name but with only the initial letter in uppercase and other letters in lowercase. For example, if a schema name CREW has been defined, the property name 'Crew' in some other schema implies the declaration 'Crew : CREW' and instantiates to the set of all instances belonging to CREW.
- iii . A single instance of the object is referred to by using the same name entirely in lowercase. For example, the property name 'crew' implies the declaration 'crew : Crew ' and instantiates to a single instance of the

² Refer to the AirLine Transportation Example Fig. 1.1.

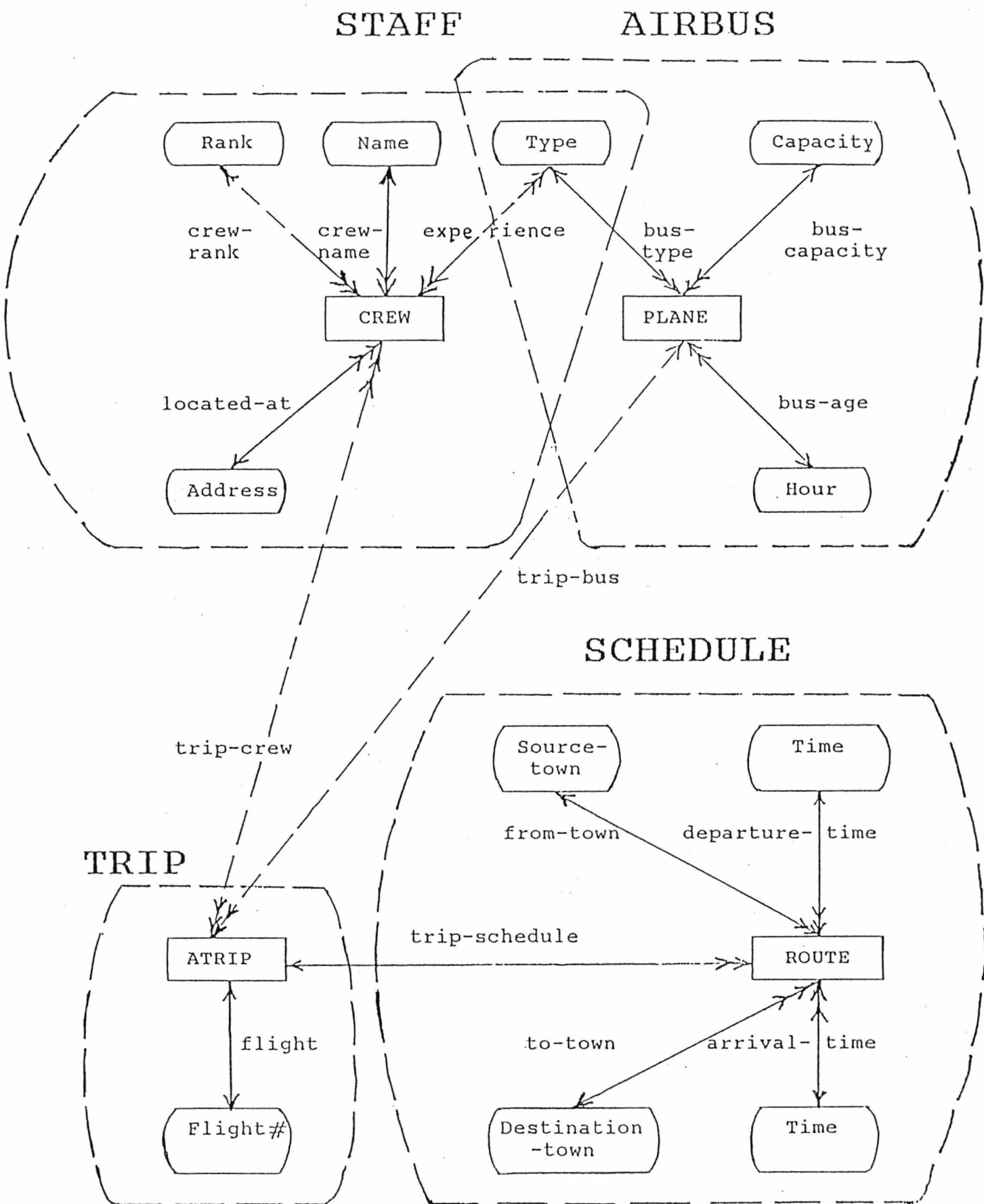


Fig.1.1. AirLine Transportation Example

Crew set.

- iv . The data properties or variables declaration in the schema have to obey the same rules. But, if a property name is not previously defined in any schema, then its type has to be declared.

A KBZ system consists of a collection of system and user-defined objects. Objects in KBZ systems are split into three different categories :

- A. Basic objects
- B. Structural objects
- C. Derived (Complex) objects

1.4.1. Basic Objects :

The basic objects are categorised into :

- entity set
- attribute set

An entity set is declared simply by naming it. For example, CREW and PLAN are entity sets. Elements of entity sets (instances) are represented internally by surrogates as shown in Fig.1.2.

On the other hand, attribute sets require schemata to define them, since their type and constraints (if any) must be declared. Fig.1.3. shows examples of attribute set definitions. The data part defines a single attribute name and the type of the attribute.

1.4.2. Structural Objects :

The structural object defines the relationships between two basic objects. For example, all relationships in the "AirLine Transportation Example" can be represented as shown in Fig.1.4.

It has to be mentioned that the relationships are defined in two directions in order that both the degree of the relationship (one-to-one, one-to-many ... etc.) and the existence characteristics (total, partial) are specified clearly and completely. The first

CREW

crew : Surrogate

PLANE

plane : Surrogate

ATRIP

atrip : Surrogate

ROUTE

route : Surrogate

Fig.1.2. Entity Set Definitions

NAME

name : String
length(name) <= 20

TYPE

type : String
length(type) <= 10

ADDRESS

address : String
length(address) <= 40

(Note: 'length' is a system function)

Fig.1.3. Attribute Set Definitions

RANK

rank : String
rank ∈ {'pilot', 'co-pilot', 'navigator', 'staff' }

CAPACITY

capacity : Integer
capacity ≤ 1000

HOUR

hour : Integer
hour ≤ 100000

FLIGHT#

flight# : Integer
flight# ≤ 100000

TIME

time : Integer
time ≤ 2400

SOURCE-TOWN

source-town : String
length(source-town) ≤ 20

DESTINATION-TOWN

destination-town : String
length(destination-town) ≤ 20

Fig.1.3. Attribute Set Definitions (continued)

CREW-NAME

```
title    : Crew ----> Name  
         : Name ----> {Crew}
```

CREW-RANK

```
ranked   : Crew ----> Rank  
rank-of  : Rank --!-> {Crew}
```

LOCATED-AT

```
location: Crew ---> Address  
occupied: Address -->{Crew}
```

EXPERIENCE

```
expertisc:Crew ----> {Type}  
         :Type --!-> {Crew}
```

BUS-TYPE

```
has-type: Plane----> Type  
type-of  : Type ---->{Plane}
```

BUS-CAPACITY

```
         : Plane----> Capacity  
         : Capacity--->{Plane}
```

BUS-AGE

```
         : Plane---> Hour  
         : Hour ---> {Plane}
```

FLIGHT

```
         :Atrip---->Flight#  
         :Flight#-->Atrip
```

FROM-TOWN

```
         : Route----> Source-town  
         : Source-town--->{Route}
```

Fig.1.4. Structural Objects Representation

TO-TOWN

```
: Route----> Destination-town
: Destination-town----> {Route}
```

DEPARTURE-TIME

```
: Route---> Time
: Time ---> {Route}
```

ARRIVAL-TIME

```
: Route---> Time
: Time ---> {Route}
```

TRIP-SCHEDULE

```
:Atrip---->{Route}
:Route----> Atrip
```

TRIP-CREW

```
: Atrip---->{Crew}
: Crew ---->{Atrip}
```

TRIP-BUS

```
: Atrip----> Plane
: Plane---->{Atrip}
```

----> ... Total Function

--|-> ... Partial Function

{ } ... Finite Set

Fig.1.4. Structural Objects Representation (continued)

relationship named is the primary relationship, while the second is the inverse. The primary and/or inverse names in the data part of a relationship schema may be omitted. For example, the inverse name in the CREW-NAME relationship is omitted. For more discussion about structural objects refer to [Oxborrow 88.a].

1.4.3. Derived (Complex) Objects:

Derived objects are constructed by including basic objects or relationships inside a new schema declaration to define a new class. Derived objects can also be based on other derived objects. The class inherits all the properties (data and functional) of the objects specified inside the schema. This provides support for rich semantics. A class may or may not contain any constraints, functional properties or predicates. Derived objects can be classified into :

A. Entity Aggregate :

Current DBMSs depend upon a simple record structure underlying the database. This simple record structure can be implemented in KBZ. Using predefined relationships, entity aggregates can be declared by listing the names of relevant relationships, together with any rules associated with attributes in the relationships. For example, Fig.1.5. illustrates entity aggregate representation. Constraints may be declared (if they exist). However, in this example, no constraint exists. An entity aggregate instance in KBZ can be mapped onto an n-tuple in a relational DBMS. Fig.1.6. describes the STAFF representation in relational DBMS. However, it is clear that the STAFF entity aggregate representation has more semantics than the STAFF relational representation.

The basic feature of entity aggregates is that the domain of all the relationships in any entity aggregate schema is the same basic object. In Fig 1.5., the domain of all the relationships of STAFF object is the entity set 'Crew'.

B. Subtypes :

The entity aggregate SCHEDULE in the example described in the Fig.1.5. illustrates all the schedules in the airline database. The overall schedule in some situations is

STAFF

crew-name crew-rank experience located-at
--

AIRBUS

bus-type bus-capacity bus-age

TRIP

flight

SCHEDULE

from-town departure-time to-town arrival-time
--

Fig.1.5. Entity Aggregate Representation

STAFF(Crew, Name, Rank, Type, Address)

Fig.1.6. STAFF Relational Representation

ENGLAND-SCHEDULE

schedule
source-town ∈ {'London', 'Gatwick'}

LONDON-SCHEDULE

schedule
source-town ∈ {'London'}

Fig.1.7. Subtype Object

not important to each branch distributed all over the world, since each branch is interested only in its own flights. That is, a category (or subtype) of the schedule is needed only. For example, LONDON-SCHEDULE, and ENGLAND-SCHEDULE can be derived from the SCHEDULE entity aggregate as shown in Fig 1.7.³

The SCHEDULE entity aggregate represents "generalization", while "ENGLAND-SCHEDULE" and "LONDON-SCHEDULE" subtype objects represent "categorization" or "specialization". "Generalization" and "specialization" define "type hierarchies". A subtype may simply inherit the properties of the parent in the hierarchy, or it may, in addition, possess its own properties.

For other types of derived objects, refer to the original KBZ paper mentioned above.

1.5. Introduction to Parallel Processing :

As mentioned before, this research is being carried out to develop a prototype Parallel KBZ (PKBZ) OODBMS. In the previous subsection an overview of the KBZ OODBMS was illustrated. In this subsection, a survey of the different parallel systems will be described.

The large development of VLSI technology has enabled and encouraged the widespread use of multiprocessor and multicomputer systems, so parallel computers become more common, and "as common computers become more parallel, it is important that the system designers and programmers cast off the shackles of sequential thought" [Hey 90].

On the other hand, most real-time system applications are inherently parallel in nature. Programming such applications in purely sequential languages on the uniprocessor machine has inevitably caused the applications to lack both the semantic aspects of real life and the expressive power to deal with the problem domain. The reasons for finding the parallel method of operation preferable to sequential mode are

³ In the example, one assumes that 'London' and 'Gatwick' are the only towns that have airports in England

quite apparent.

1.5.1. Classification of Computer Systems :

Computer systems may be classified into a number of collective groups. Each group is determined by the type of processing which is required, together with the method by which the processing elements communicate, and the use of memory. So, all computer systems can be divided into the following categories [Flynn 86] :

A. Single Instruction Single Data (SISD) :

In this type, the computer executes instructions sequentially and may overlap (pipeline) execution. In overlap, the conventional hardware is altered in order that two or more major components may overlap operations. For example, the "Control Unit" may fetch the next instruction while the "Arithmetical and Logical Unit" is executing the current instruction. This is largely implemented using a system of pipelines and buffers. The pipeline breaks a sequence into suboperations in order that each operation is carried out simultaneously. The concept of a buffer enables decoupling to be achieved by providing a place for information to be stored. This category describes most computer systems available today.

B. Single Instruction Multiple Data (SIMD) :

These are known as vector or array processors, and are often microprocessor based designs with multiple processing elements, a single Central Processing Unit and limited memory. In SIMD, several processors execute the same instruction but on different data originating from the memory associated with each processor. The DAP (Distributed Array Processor) [Hockney 81] is an example of SIMD. It consists of an array of processing elements working in parallel.

C. Multiple Instruction Single Data (MISD) :

Here, although there is only one data stream, different parts of different instructions are being processed at the same time, this is the standard pipeline

processing found in machines such as the IBM 3083 and ICL 3900 series [Cook].

D. Multiple Instruction Multiple Data (MIMD) :

This implies that the distinct threads of control can be executing different instructions and manipulating different data structures. This category covers most multiprocessor and tightly coupled parallel processor computer systems. Tightly coupled is the situation in which there is a great deal of processor interaction via shared memory which may be logically addressed and directly accessed by all processors. A network of transputers is an example of MIMD. But, each transputer has its own local memory. That is, the memory is attached to each processor. The architecture of the transputer will be discussed in the next subsection.

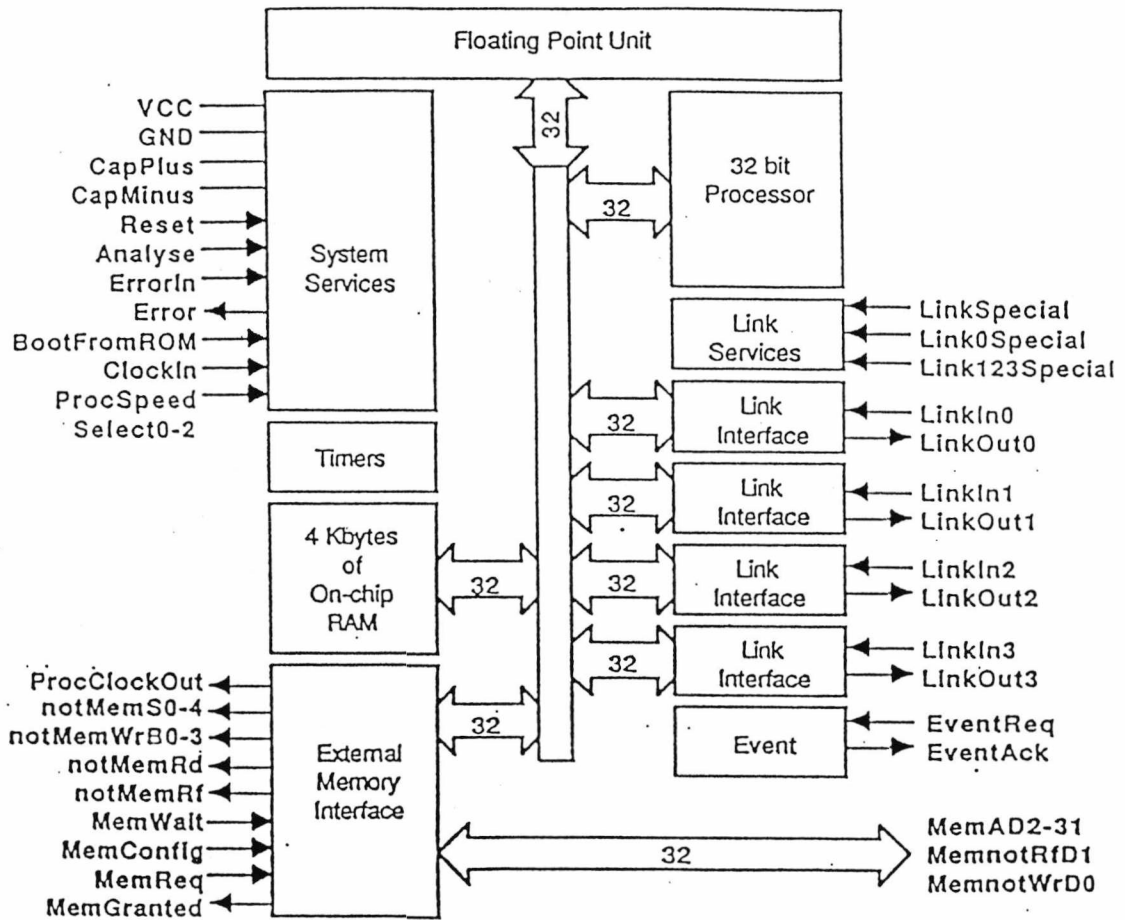
1.6. Hardware Architecture of The Transputer :

One development in the parallel processing area is provided by a new computer architecture developed by Inmos Ltd as a microprocessor. This new processor is called a transputer (TRANSistor comPUTER). The first member of the Inmos transputer family, the IMS T414, was first announced in 1983. A typical member of the transputer product family is a single chip containing processor, memory, and communication links which provide point to point connection between transputers. Fig.1.8. shows the transputer hardware architecture. Communication across the link takes place only once both ends are ready, so that events are synchronised. The synchronisation of events was one of the major problems in early attempts at parallel processing.

Current transputer products include the 16-bit IMS T212, the 32-bit IMS T414 and the IMS T800. IMS T800 is a 32-bit transputer similar to the IMS T414, but with an integral high speed floating point processor [Inmos 87][Inmos 88.a].

A transputer can be used in a single processor system or in networks to build high performance concurrent systems, the system performance depending on the number of transputers, the speed of inter-transputer communication and the floating point performance of each transputer.

A network of transputers is easily constructed using point-to-point



IMS T800 Block Diagram

Fig.1.8. Transputer Hardware Architecture

communication. That is, the transputers are hard-wired together in a physically rigid structure. The processing elements are homogenous and require no control, each cell performing one computation at each step of the process, with input and output being overlapped with the computation so providing a high performance. The basic transputer processor speed is in the region of 10 MIPs, but because transputers do not share the same communication bus the overall processing power increases theoretically linearly with the number of transputers added; an array of, say, 32 such transputers should provide a speed in the region of 320 MIPs. In the case of conventional processors, the overall processing power improvement starts to diminish with the involvement of around six processors [Brookes 89].

The actual physical connections among the cells will depend upon the specific problem for which the architecture is intended to be used. It has been found, as stated in [Carling 88] that certain interconnection patterns are more efficient when applied to the structure of a specific problem. Because the systolic architectures under discussion are hard-wired, the actual configuration can be optimised for one particular algorithm by selecting a specific connection pattern.

1.7. Occam and The Transputer :

Transputers can be programmed in many high level languages, and are designed to ensure that compiled programs will be efficient. Where it is required to exploit concurrency, but still to use standard languages, Occam [Jones 88][Brookes 89] can be used as a harness to link modules written in the selected languages. For example, transputers can be programmed in C, Fortran, Pascal, Modula 2 and Ada, all with programming extensions allowing programmers to prepare code in explicitly declared segments for concurrent execution.

Occam language design has been closely associated with the development of the transputer. Although Occam is not an assembly language, the architecture of the transputer so closely implements Occam's constructs, and Occam so completely provides for control of the hardware, that no assembler is required or desired for virtually any

application.

To gain most benefit from the transputer architecture, the whole system can be programmed in Occam. This provides all the advantages of high level language, a maximum program efficiency and the ability to use the special features of the transputer.

Occam is an abstract language that has the dual role of being an implementation language and a design formalism. A system is designed in terms of an interconnected set of blocks. Each block can be considered as independent process. Thus, each process can be regarded as an independent unit of design. Processes are connected to form concurrent systems. Each process can be regarded as a black box with internal state, which communicates with other processes. Processes can be used to represent the behaviour of many things, for example, a logic gate, a microprocessor, a machine tool or an office.

A process communicates with other processes along point-to-point channels. Its internal design is hidden, and it is completely specified by the messages it sends and receives. Communication between processes is synchronized, removing the need for any separate synchronization mechanism. Internally, each process can be designed as a set of communicating processes. The system design is therefore hierarchically structured. At any level of design, the designer is concerned only with a small and manageable set of processes. The Occam language is based on these concepts, and supports the transputer architecture from the logical point of view.

The processes themselves are finite. Each process starts, performs a number of actions and then terminates. An action may be a set of sequential processes performed one after another, as in a conventional programming language, or a set of parallel processes to be performed at the same time as one another. Since a process is itself composed of processes, some of which may be executed in parallel, a process may contain any amount of internal concurrency, and this may change with time as processes start and terminate.

As a result, Occam can be used to program an individual transputer or to program a network of transputers. When Occam is used to program an individual transputer, the

transputer shares its time between the concurrent processes and channel communication is implemented by moving data within the memory. When Occam is used to program a network of transputers, each transputer executes the process allocated to it. Communication among Occam processes on different transputers is implemented directly by transputer links. Thus, the same Occam program can be implemented on a variety of transputer configurations, with one configuration optimized for cost, another for performance, or another for an appropriate balance of cost and performance.

1.8. Research Objectives :

There are many researches which have designed different system applications using parallel processing techniques. There are also many research efforts that design and implement OODBMSs, but none of these OODBMSs, according to the author's knowledge, is designed or implemented using parallel processing techniques. That is why this research is being carried out.

Thus, the main research objective is to explore parallel processing in an object oriented database management system. So, the main tasks of this research are involved in :

A. Investigation of The Different Aspects of Parallelism :

As mentioned before, object oriented database management systems are intended to meet the needs of new and emerging database applications. But, an OODBMS is a complex system. Moreover, a number of hard problems remain to be solved. These include : improving the overall performance of object oriented system. As the improvement and widespread use of parallel computers become more common, it is important to design and implement a complex system such as OODBMS using parallel processing. This is due to :

1. Most real systems are inherently parallel in nature. So, the implementation of these systems in sequential manner must inevitably lack some of the semantic aspects of the real problem.

2. Development of OODBMS is expected to solve both the efficiency and cost performance problems which may be encountered in uniprocessor systems.

The main mechanism for parallel processing, in most of the current applications, is to distribute the work load of the system across different processors simultaneously to gain the high performance of the system. However, this is not the only mechanism for parallel processing. In this research, other mechanisms for parallel processing with regard to OODBMSs are introduced to increase the system performance.

B. Development of An Experimental Parallel OODBMS :

KBZ [Oxborrow 88.a] is taken into consideration as a specific example to build an OODBMS prototype in this research. The development of the experimental KBZ OODBMS is used as a test-bed for many of the aspects of parallelism which will be discussed.

Moreover, the system prototype is implemented to satisfy the "Parallel Processing Transparency" concept. That is, the end user must be unaware of how his own message would be executed in parallel in different processors.

It has to be mentioned that the prototype is implemented on the Transputer Computing Surface at the University of Kent using Occam programming language. The main reasons for using Occam are :

1. Occam model is simple to simulate the "real world" concept. The "real world" is overflowing with concurrent objects and Occam has the capability to capture their structures directly down to a fine level of granularity. It is simple to model an object's behaviour, internal data structures and interactions with other objects from its own point of view [Welch 88]. Moreover, Occam does provide support for sophisticated message structures through its concept of PROTOCOL. Its model of message-passing parallel processes forms the basis for more powerful and explicit abstractions of the "object" concept in the object oriented paradigm.

2. Occam system language is a simple language founded upon a simple and secure model of concurrency for transputers. Moreover, Occam has a small range of constructs.

1.9. Thesis Layout :

This thesis consists of seven chapters. In this chapter, the research background

has been introduced. It contains, the features and limitations of the current traditional database management systems, the new features of non-traditional database management systems, the potential of the object oriented approach, the potential of parallel processing and the research objectives.

In chapter two, a survey of the state of the art in related researches is given.

In chapter three, one of the main objectives of this research is illustrated. This chapter is concerned with the different aspects of parallelism which can be implemented within the scope of the object oriented database management systems.

In chapters four and five, the second objective of this research is discussed; that is, the development of an experimental KBZ OODBMS prototype on the transputer Meiko Computing Surface.

In chapter six, both the enhancement of the prototype and the experimental results are discussed.

In chapter seven, the conclusions and the future researches are described.

CHAPTER 2

The State of The Art In Related Research

2.1. Introduction :

In this chapter, the state of the art in related research will be discussed. Since, this research relates to two major research areas, a small selected number of existing systems which cover both the object oriented paradigm and parallel processing techniques are investigated. Two sequential OODBMSs will be discussed: EXODUS [Carey 86.a], and IRIS [Fishman 87]. In addition, object oriented systems with parallel features will be illustrated: PRESTO [Bershad 88], Rekursiv [Harland 88], and CHANCER [Chalmers 89]. Finally, an application of parallel query processing on a Meiko Computing surface will be discussed [Kerridge 87].

2.2. Object Oriented Database Management Systems :

2.2.1. EXODUS DBMS:

EXODUS [Carey 86.a] is a DBMS that facilitates the fast development of high-performance, application-specific database systems. EXODUS provides certain kernel facilities, including a versatile storage manager and a type manager. EXODUS has been designed at the University of Wisconsin.

A. An Overall EXODUS System Architecture :

EXODUS is designed to provide a toolbox that can be easily adapted to satisfy the needs of new applications such as engineering design, image and voice data management ... etc. EXODUS supplies at its lowest level a layer of software termed the "Storage Object Manager" which provides support for concurrent and recoverable operations on arbitrary size storage objects.

EXODUS provides either a "generator" or a "library" to aid the user in generating the appropriate software. EXODUS is expected to be used for a wide variety of

applications, each with a potentially different query language. As a result, it is not possible for EXODUS to furnish a single generic query language, and it is accordingly impossible for a single query optimizer to suffice for all applications. Instead, EXODUS provides a generator for producing query optimizer for algebraic languages.

Fig.2.1. represents the structure of an application-specific database management system implemented using EXODUS. The Storage Object Manager provides capabilities for reading, writing, and updating "storage objects". The storage object is an untyped, uninterpreted variable-length byte sequence of arbitrary size. The Storage Object Manager is enhanced by providing buffer management, concurrency control, and recovery mechanisms for operations on shared storage objects.

The Storage Object Manager contains the E programming language and compiler. E extends C by adding the notion of persistent objects to the language's type definition. Whenever persistent objects are referenced, the E translator is responsible for adding the appropriate call to fix/unfix buffers, read/write the appropriate piece of the underlying storage object, lock/unlock objects, ... etc. Thus, the user is freed from having to worry about the internal structure of persistent objects. For buffering, concurrency control and recovery, the E language includes statements for associating locking, buffering, and recovery protocols with variables that reference persistent objects. The objective of E is to simplify the development of internal systems software for a DBMS.

A collection of access methods is layered above the Storage Object Manager. Access methods provide associative access to files of storage objects. For access methods, EXODUS provides a library of type-independent index structures including B+ trees and linear hashing.

The "Operator Methods" layer contains a mix of user's code and EXODUS code. This layer contains a collection of methods that can be combined with one another in order to operate on storage objects.

The "Type Manager" provides schema support for a wide variety of application-specific database systems. The data modelling facilities provided by the "Type Manager"

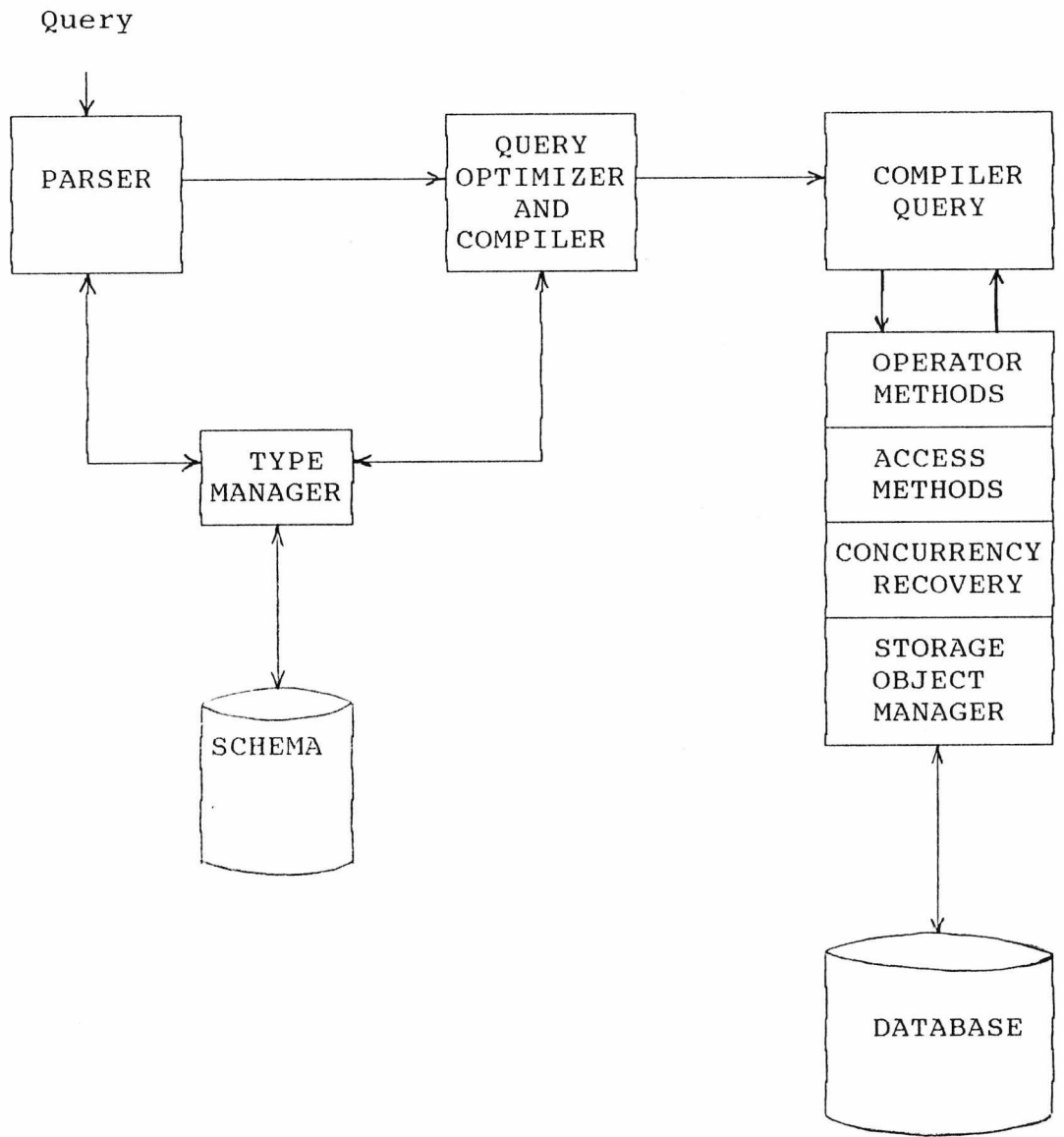


Fig.2.1. The Structure of An Application-Specific DBMS Implemented Using EXODUS

are basically those of a generalized class hierarchy with multiple inheritance.

To start an EXODUS operation, a query is sent to the "PARSER". The parser is responsible for transforming the query from its initial form into an initial tree of database operators. After parsing, the query is optimized, and then compiled into an executable form. During the parsing and optimization phase, the "Type Manager" is invoked to extract the necessary schema information.

B. EXODUS Conclusion :

EXODUS is an extensible database system which is intended to simplify the development of high-performance, application-specific database systems.

EXODUS includes two components that require little or no change from application to application - the Storage Object Manager and Type Manager. The Storage Object Manager in EXODUS is a flexible storage manager that provides concurrent and recoverable access to storage objects of arbitrary size. The Type Manager is a class-based schema management subsystem. The base types can be extended by the user. In addition, EXODUS provides libraries of database system components that are likely to be widely applicable, including components for access methods, version management, and simple operations. Furthermore, EXODUS provides the E database implementation language which supports persistence and to a great extent shields the user from the recovery protocols.

2.2.2. Iris DBMS :

The Iris [Fishman 87][Fishman 89] database management system is a research prototype of an OODBMS being developed at Hewlett-Packard laboratories. Iris is intended to meet the needs of new and emerging database applications such as office information and knowledge-based systems, engineering test and measurement, and hardware and software design.

A. Iris System Structure :

Fig.2.2. shows the layered architecture of Iris. The Object Manager implements the

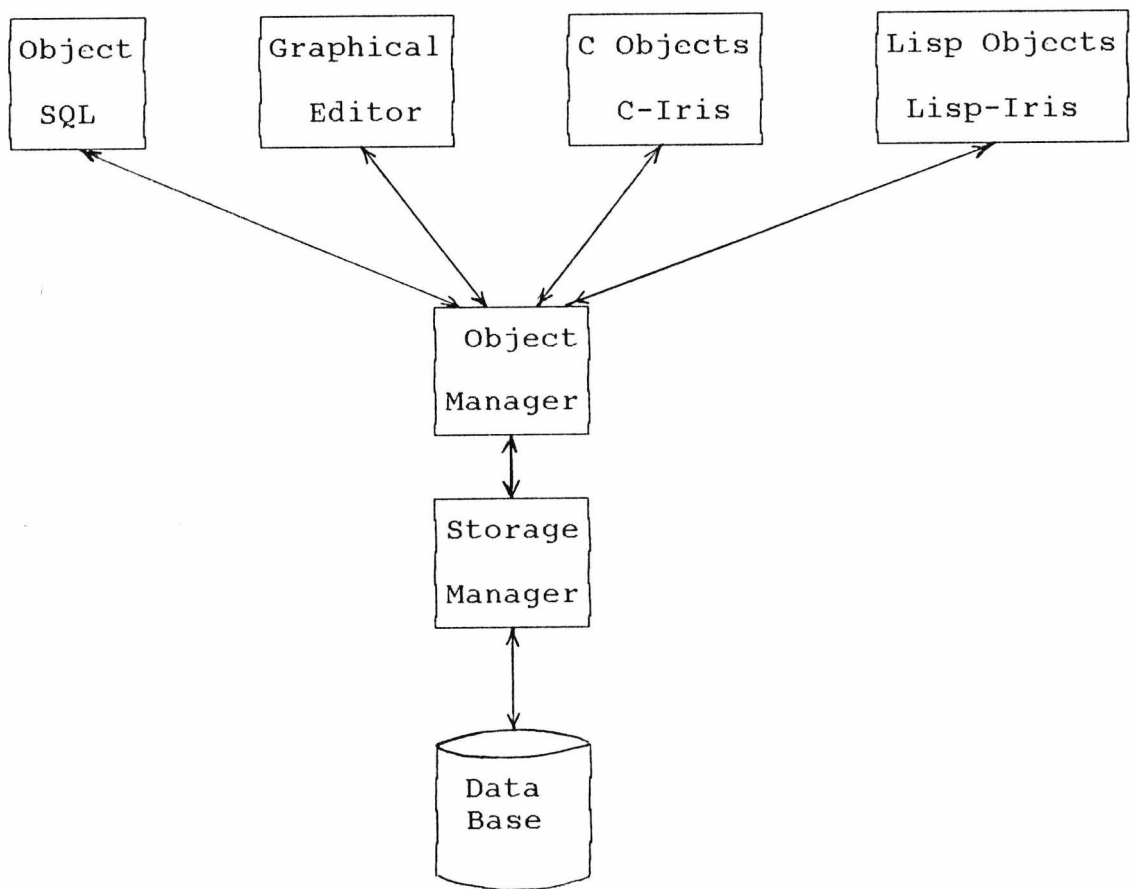


Fig.2.2. Iris DBMS Structure

Iris data model which supports high-level structural abstractions such as generalization/specialization, and aggregation. The query processor of Iris depends upon an extended relational algebra format. So, the query is optimized and then interpreted against the stored database.

The Iris Storage manager is a conventional relational storage subsystem. It provides associative access and update capabilities to a single relation at a time and includes transaction support.

Iris is accessible via interactive interfaces or through interface modules embedded in programming languages, such C-Iris and Lisp-Iris in Fig.2.2.

There are two interactive interfaces, Object SQL (OSQL) which is an object-oriented extension to SQL, and a Graphical Editor which allows the user to interactively explore the Iris type structure as well as the inter object relationship structures defined on a given Iris database.

B. Iris Conclusion :

Iris is an OODBMS prototype. It is implemented in C. The Storage Manager still essentially needs modification, since it is implemented by a conventional relational storage subsystem. Other object-oriented programming languages, including C++ are under investigation for use in redesigning the Iris System.

2.3. Object Oriented Systems with Parallel Features :

2.3.1. PRESTO :

PRESTO [Bershad 88] is a programming system for writing object-oriented parallel programs in a multiprocessor environment. PRESTO consists of an object-oriented language C++ [Stroustrup 86], a library of basic tools constructed in this language, a run-time system providing efficient support, and a programming methodology.

PRESTO is designed and implemented to build distributed object-oriented systems in a multiprocessor environment. In distributed systems, an object-oriented programming paradigm makes it easier to think about and to express concurrent algorithms. Each

object is responsible for maintaining its own internal consistency.

Moreover, PRESTO provides efficient concurrency and synchronization mechanisms. PRESTO allows the programmer to use parallelism in the manner most natural to the problem at hand.

A. Why C++ with PRESTO:

PRESTO is implemented in C++ because C++ supports the object-oriented paradigm. PRESTO currently runs on the Sequent shared-memory multiprocessor on top of the Dynix operating system. That is, PRESTO exists now on only one machine. But, it can be ported to other multiprocessors, since it is written in a high level language.

Since PRESTO is written in C++, it is most naturally used with applications written in that language. Although it is possible to use the system from other languages, many of PRESTO's concepts will be difficult and time-consuming to express. So, users are encouraged either to write completely in C++, or to build application-specific interfaces between languages.

B. PRESTO object model :

PRESTO provides the programmer with several classes useful for writing parallel programs. A "class" in PRESTO is a user-defined data type allowing the programmer to specify an object in terms of its data representation and operations. These classes, and the environment in which they execute, help support two of the major goals of PRESTO: efficient execution and comfortable abstractions for expressing concurrency.

In PRESTO, all objects execute in a single address space shared by all processors, allowing for fast inter-object communication and synchronization through shared storage. In a sequential object-oriented system, an object hides its data and its implementation. In PRESTO, an object hides not only its data and its implementation, but also its execution. That is, when a caller invokes an operation on an object, the caller is unaware whether that operation executes sequentially or in parallel. The implementor of an object determines the extent of parallelism that is appropriate to the object. Dealing with concurrency in this manner simplifies the task of writing parallel

programs.

C. PRESTO Thread Class :

Thread objects (threads) are the building blocks of PRESTO parallel programs. There are two essential operations that can be performed on a thread. A thread can be "created", allowing the creator to specify the thread's qualities, such as its name and maximum storage requirements. Once created, a thread can be started executing some operation of some object, where in it executes in parallel with the starting thread. "Start", in fact, is an operation defined for threads; parameters include the object, the operation where the thread is to be started, and any parameters expected by that operation.

The "user" of an object chooses between synchronous and asynchronous invocations, and the "implementor" of an object chooses between sequential and parallel execution.

An object cannot tell whether it is being invoked synchronously or asynchronously, and the user of an object cannot tell whether an invocation is being performed sequentially or in parallel.

Each thread has its own call-stack. Any objects declared on that call-stack are visible only from within the thread to which the call-stack belongs. If data is to be shared between threads, then it should be declared as such within the object's definition.

D. PRESTO System Architecture :

PRESTO exists as a run-time library on a Sequent shared-memory multiprocessor. The Sequent's operating system is Dynix which supports shared memory. Dynix provides support for writing parallel programs, but this support is limited. Because the basic synchronization mechanisms are cumbersome to use, a "parallel programming library" is provided. This library restricts the "threadedness" of a parallel program to the number of physical processors in the system.

The basic role of the PRESTO run-time system is to map users' threads onto

physical processors and to provide access to a global shared memory in which all objects reside. PRESTO maps threads onto Dynix processes, relying on the Dynix kernel to complete the mapping onto a physical processor.

The PRESTO system maintains a single scheduler object. The scheduler object keeps track of all threads. Each processor in the system is represented by a processor object. One scheduler thread runs within each processor object, and that thread's only activity is to request threads from the scheduler object.

A thread may execute on different processors at different times. Migration occurs only if a thread is blocked and then resumed at some later time when some other processor is idle. Scheduler threads are an exception to this; they never migrate. A scheduler thread runs only on processor for which it is scheduling. Fig. 2.3. shows how the scheduler object, processor objects and physical CPUs are related. Because these objects interact with one another only through their operations, each can be easily replaced or modified without affecting the others. For example, the scheduler object could be changed to maintain multiple priority queues for threads rather than a single runnable queue. Since scheduler threads interact with the scheduler only through a "GetAThread()" operation, they would remain unaffected by the change.

The PRESTO scheduler eventually halts when there are no longer threads. At this point, all existing synchronization objects are destroyed. If any one of them indicates a waiting thread, the system declares deadlock and displays the state of all interminably blocked threads.

E. PRESTO Conclusions :

PRESTO is a good system in which it joins the classical notions of concurrent programming with the powerful concepts of object-oriented design. Objects can be made completely responsible for their own execution, as well as modification and presentation.

PRESTO is the current system of choice for parallel programming at the University of Washington. Certain applications have been built using PRESTO. For example, a parallel solution package for queuing network performance models is built

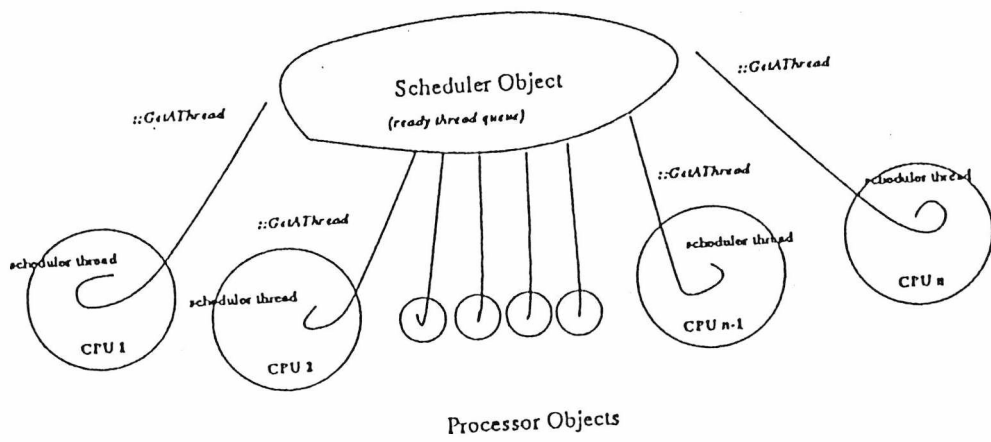


Fig.2.3. Relation Among Scheduler Object, Processor Object and Physical CPU in PRESTO

using PRESTO.

However, as indicated before, the synchronization mechanisms among objects are cumbersome and costly to use in a parallel program. Moreover, a great effort has to be made in writing software to perform such synchronization. That is why, a system library in Dynix provides support for writing parallel programs, but this support is limited as indicated in [Bershad 88]. These synchronization mechanisms are solved in Occam using a transputer system, since the messages among transputers are synchronized naturally.

Moreover, in PRESTO, all objects execute in a single address space shared by all processors. Such a single shared memory address space will lead to slow performance among objects. That is because only a single address is performed at a time. However, this problem can be easily solved using Occam on the transputer system, since each transputer has its own local primary memory. So, more than one address space can be accessed simultaneously.

2.3.2. Rekursiv : An Object-Oriented CPU:

In 1984, Linn Smart Computing announced the Rekursiv [Pountain 88][Harland 86][Harland 87][Harland 88]. Rekursiv is an innovative new computer architecture designed around principles of object-oriented programming. Rekursiv is designed as a chip set with which to implement a persistent-store, object-oriented processor.

Persistence and object orientation are natural partners, because if a program is a simulation built from objects, then these objects must be expected to live for as long as their real world counterparts.

Rekursiv is designed to allow very high-level instructions. The very name Rekursiv suggests that machine instructions can be made arbitrarily complex, including recursive calls and even calls to other programs; for example, a tree-walking routine can be microcoded as a single instruction.

A. Rekursiv Architecture :

Rekursiv is a singleboard rather than a single chip microprocessor. Rekursiv

achieves high performance by having multiple internal memory buses so that many operations can occur in parallel.

Rekursiv is built from three custom gate arrays and several megabytes of fast static RAM. Fig.2.4. shows a block diagram of the main functional units of Rekursiv, where the three gate arrays are the blocks called "Objekt", "Numerik", and "Logik".

The SRAM holds the microcode and the pager tables used to keep track of objects; it exists inside the processor. The SRAM is organized into six different functional memory spaces, each with its own data and address buses. Since, there is also a dynamic RAM (DRAM) interface for the main object store memory, the Rekursiv could be labelled a seven-memory architecture.

The six blocks that are implemented in SRAM are the two stacks (control and evaluation), "control store", and "control store map", the "pager tables", and the block marked "NAM".

Most of Rekursiv's internal data paths are 40-bits wide, though the DRAM address bus is only 24-bits wide. Objects are stored both in DRAM and on hard disk, in what could be thought of as equivalent to the external memory of a conventional processor, but since the microcode can access this memory, too, the distinction between inside and outside is blurred.

B. Rekursiv Operation :

The main advantage of Rekursiv is that it combines the functions of a CPU, memory manager, database manager, and operating system all in one. During the execution, the Rekursiv creates objects, pages them back and forth between memory and hard disk, and performs arithmetical and logical operations on the data in their fields.

An object, in Rekursiv, can be considered as just a chunk of memory divided into fields that holds its data and represents the "instance variables", in Smalltalk parlance.

Every Rekursiv object is defined by a unique 40-bit number that is assigned to it at its creation (from a counter called the allocator) and that remains with it for

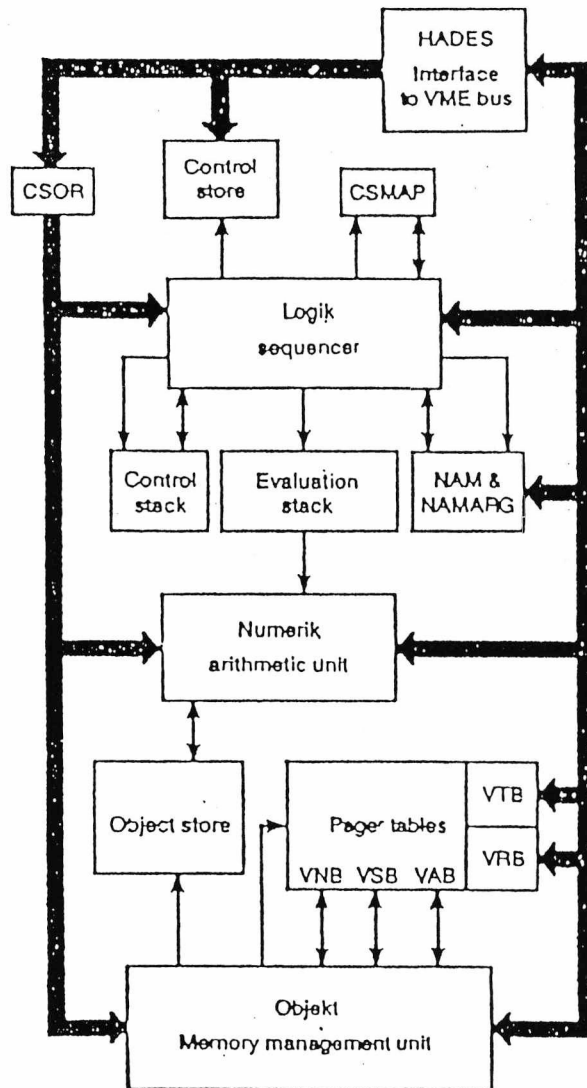


Fig.2.4. A Block Diagram of Rekursiv

its life. This number is the only way of referencing the object, because only the processor knows its real address.

Objects are stored on the secondary storage with a prefixed header. This header holds the object's number, size, and type. Type is stored in a 40-bit value that the software must interpret in some meaningful way, since types are objects too. When an object is transferred into memory, only the data fields are written into object store; the header is stripped off and written to a slot in the pager table along with the new address of the data in main memory. The pager table contains 65,536 (i.e. 64 K) slots that it can address very quickly by hashing on the object number.

If the pager table slot for a requested object is occupied, then the object's address is retrieved from the slot along with the offsets to index its fields, and the requested operation is performed on its data. During this table look up, the hardware also checks the type field, refuses to perform operations that are not allowed on the type and performs range checks so that a field, that lies outside the object, may not be indexed.

If the slot for the requested object is empty, that means it is not currently in main memory and there is a page fault. The Rekursiv is stopped dead while a signal is sent to an external disk processor to fetch the object into memory.

This disk processor has its own B-tree directory structure, which allows it to find an object's image on disk through its number. When the object has been fetched to memory and its header put in the pager table, the Rekursiv resumes processing as if nothing had happened, without any need to restart the current instruction, as there would be with a conventional processor. Page fault recovery occurs "below" the level of instruction execution, rather than being an external operating-system task. It is this property that enables Rekursiv's microcoded instructions to be of arbitrary complexity and to include recursive calls that are forbidden to normal CPUs.

The programmer's view of Rekursiv is of a truly object-oriented processor in which there is no concept of an address, only of object numbers, and where objects persist until they are destroyed. Programs can be executed only by requesting an

operation on a numbered object.

C. Rekursiv Microcode :

Rekursiv has a writable instruction store, it will come out of the box absolutely empty and is incapable of doing anything at all until the microcode for an instruction set is loaded into the control store part of the SRAM.

A standard instruction set that supports "C", together with corresponding "C" compiler, is supplied for writing application programs. Far more interesting, though, is a second instruction set, a microcoded Smalltalk interpreter, that makes far better use of the Rekursiv's unique features.

The microcode is stored in the "control store", an SRAM area with its own 16-bit bus, where there is room for 16,384 control words of 128 bits each. Another separate memory space, called the "control store map", holds a table of 2048 microcode start addresses, and maps 10-bit op codes onto the microcode that implements them. This function is equivalent to the instruction decoder of a conventional processor that would normally be hardwired logic. The control store and map can be thought of as a Smalltalk-style byte code interpreter implemented in hardware.

The op codes can be contained in objects and thus can be stored in the main object store, from which they must be fetched as in a conventional processor.

The block called "NAM & NAMARG" in Fig.2.4. is the New language Abstract Memory that stores up to 524,288 words of 10-bit op codes and their 30-bit arguments that form abstract or high-level instructions. Since the NAM is inside the processor, it behaves like a fast instruction cache.

Methods for user-defined classes would be contained in objects in the main memory and would thus incur a memory fetch, though important code can be "frozen" into "NAM".

D. Rekursiv Applications :

Linn plans to use the Rekursiv architecture in several products : an accelerator board for existing engineering workstations like those from Sun, Apollo, or Micro VAX;

a networked Rekursiv-based workstation; and the full multistation control system for flexible manufacturing that was the original goal of the Rekursiv project.

E. Rekursiv Conclusion :

The Rekursiv is an object-oriented database engine for creating and managing persistent objects and, with strict type-checking, performing just about any operation on them.

Rekursiv cannot process two or more objects in parallel, since all objects are stored in the same main memory. But, as indicated before, Rekursiv achieves high performance by having multiple internal memory buses so that many operations can occur in parallel.

2.3.3. CHANCER:

During the development of a ray tracer on a Meiko Computing Surface, problems with poor flexibility of configuration and slow software development were encountered. In order to overcome these difficulties, and in order to facilitate experimental programming on the Meiko, the Chancer system [Chalmers 89] has been built so that it can support an object-oriented style for Occam programming.

The aim of Chancer is to create a set of library modules that allow user code to be quickly developed and integrated into programs, to support better debugging facilities, and to allow program design to be based on a more flexible and dynamic model of concurrency.

Chancer supports the development of programs made up of a number of concurrently executing objects which communicate primarily by message passing. The system has been a boon to ongoing experiments in image synthesis.

A. Chancer Message :

A simple message format is used by all objects in Chancer. Messages have a header composed of five integers : "to", the id of the intended receiver, "sel", the method selector which identifies the service requested of the receiver; "from", the id of the

sender; "replyDest", the id of the object to which replies should be sent; and "len", the length of the tail array in bytes. The main use of the "replyDest" field is to nominate replacement objects during a dialogue, so spreading the workload or referring work to another object. This allows any subsequent response to be sent to the original sender without retracing the path of referrals.

A "messenger" process on each processor acts as the local agent for a "mail service". Application code resides in the processors which are connected to the messenger, i.e. inside "objects". Each messenger has an array of "slots" for such connections, and the array index combined with the processor number define the id of each object. Inter-processor routing and local delivery are the responsibility of the messenger. Objects take on the responsibility of conforming to the protocol for message delivery and transmission. By using the message system one can design an object in a manner independent of the locations of the objects it communicates with. One need not modify communications code when program configuration changes.

B. Chancer Object :

A queue handler, Q, serves to buffer messages passing between the "object" and the "messenger", as sketched in Fig.2.5. Objects can use selectivity of message acceptance in order to ensure internal data consistency and correct behaviour transitions. The queue handler will pass in messages which match specifications of both message sender and message selector. The user can send messages to the queue handler of an object and request that reports on the state of the queue handler be sent to the screen.

Objects can be constructed according to the format shown in Fig.2.6. The process directly connected to the queue handler implements the most specialized member of the classes which make up an object. Other classes higher in the superclass hierarchy are nested inside that process. That is, Fig.2.6. shows the structure of an instance of class "C" with superclass S. C communicates with the message queue handler via channels "toQ" and "fromQ". The process N does the "normal" work for the class. Nesting of similarly structured superclasses continues until the class Object, which has no superclass, is

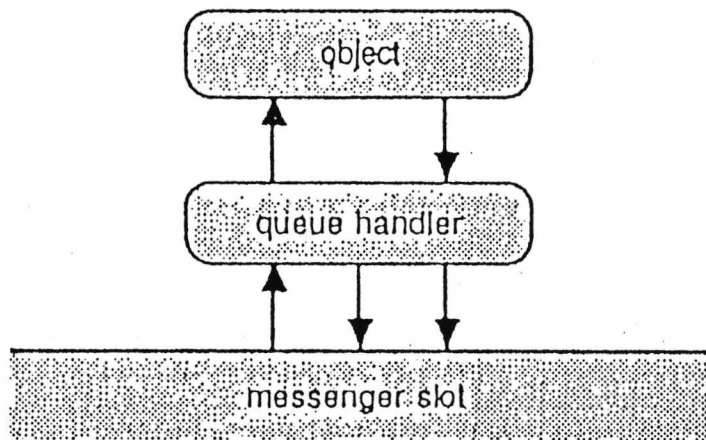


Fig.2.5. Message Buffer Handler in CHANCER

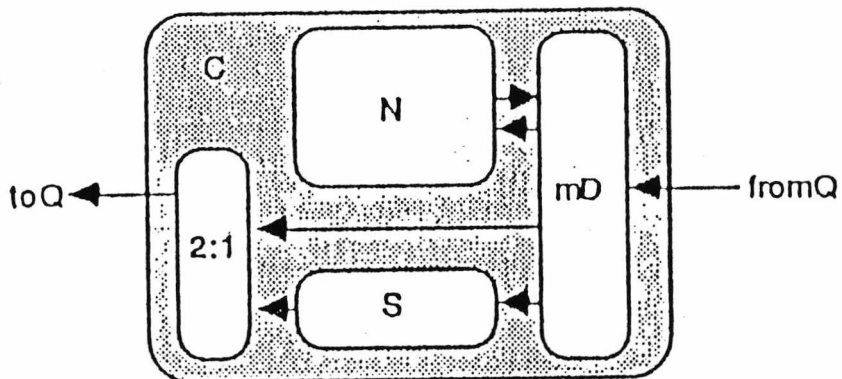


Fig.2.6. CHANCER Object Operation

reached.

A message sent in from the queue handler is passed to the "message director", mD, which performs the validation check. Messages output from N and S are taken out of the object via the 2:1 multiplexer.

The nesting of classes means that a number of N processes execute concurrently. An incoming message will be taken by the most specialized member of the classes in the object whose "message director" mD deems the message to be acceptable. This "single activation rule" ensures correct communication with the queue handler and allows a subclass to override a method of a superclass in a manner which minimises code interdependence.

C. System Operation :

An object called "monitor" reads lines of text from the input keyboard. Then, monitor transforms this text into messages and passes on via the messenger. This provides the user with access to all objects. The user now interacts with the system directly, since what is typed is, conceptually speaking, sent straight into the system.

When a program starts up, objects can be booted in several ways. From the keyboard the user can send messages to initiate activity. Alternatively, a list of objects to boot up can be included in the start up sequence for the messenger on each processor. A third method involves processes which can call one of the procedures which implement objects.

D. Chancer Conclusion :

Chancer supports the development of programs made up of a number of concurrently executing objects which communicate primarily by message passing. A global communications schema based on message transmission supports this development. The resulting programs are easily reconfigurable and code is straightforwardly reused and expanded.

2.4. Parallel Query Operation :

[Kerridge 87] represents a database machine which is capable of supporting the parallel operation of the relational operators. The database machine can support single user and multiple user queries.

The basic structure of a database machine is shown in Fig.2.7. A user process is executed in one of user processors U. The user processor U generates queries for the database which are passed to a query decomposition processor Q. Then, the decomposed query is passed to the data dictionary processor D. The query decomposition process establishes which entities are required and the operations that have to be carried out upon the entities.

The data dictionary determines which of the entity processors E_i is required and the required relational operator(s). Then, the necessary processes can be loaded into one or more of the operations processors J_k . Moreover, correct buffer processors B will be set up too.

When the desired query needs part of the complete entity, the tuples of an entity can be passed through a selection process which resides in the appropriate processor S. The data dictionary processor D will communicate the details of the selection constraint to the processor S.

A user may insert new data into the database. So, the user U may pass the data via one of the Insertion processors I to the appropriate entities E. This insertion mechanism is controlled by the link switch C004. Moreover, the link switch connects user processes U to appropriate J processor that has been allocated to the user process by the data dictionary processor D.

A. General operation :

The communications processors C have been included so that a number of such entity/user groups may be connected together to form a larger database machine. The processor C will enable the transfer of data between groups. Each group's data dictionary has to have a sufficient details of entities located in other groups to

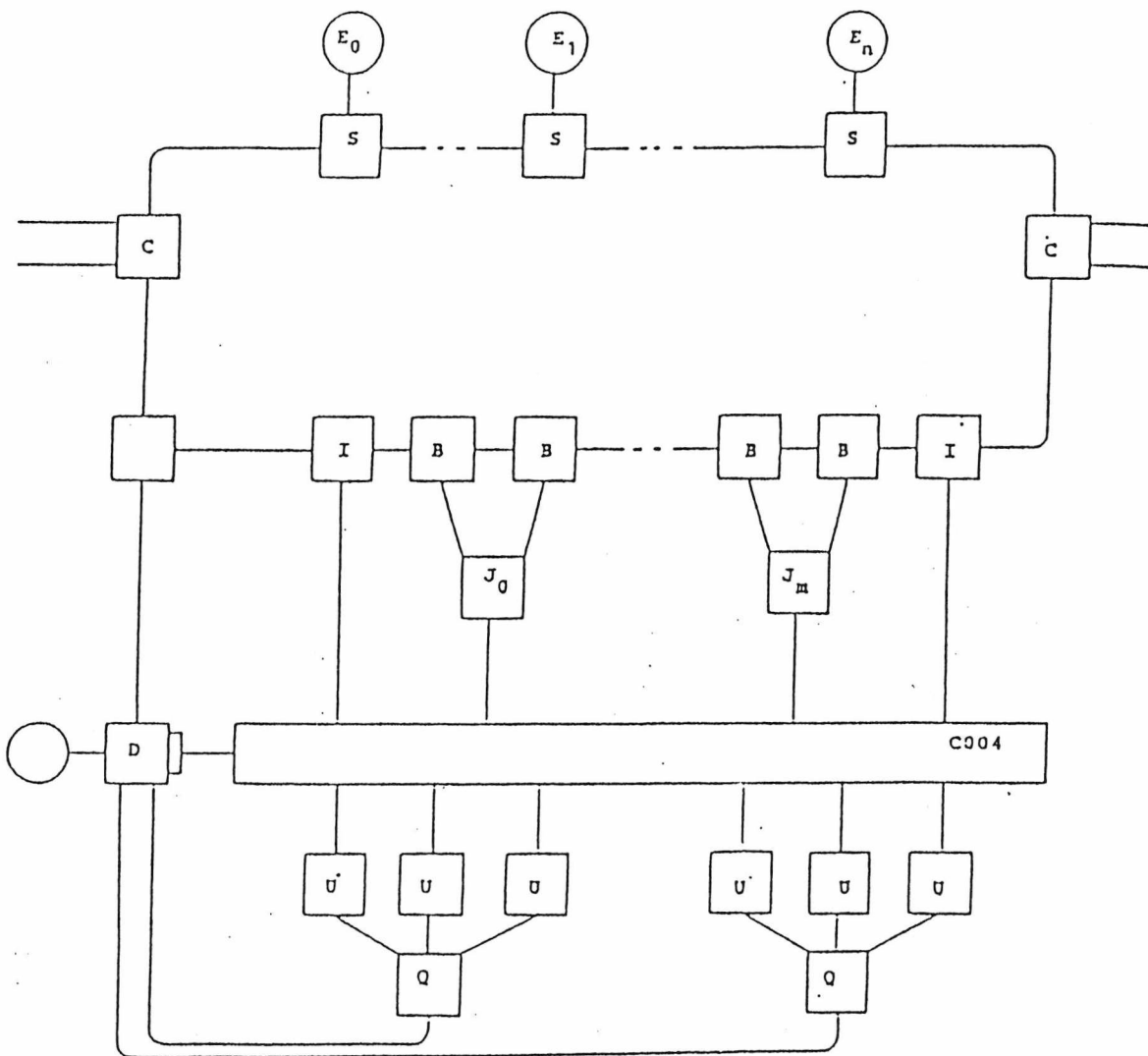


Fig.2.7. The Architecture of Parallel Query Database Machine

enable a request for remote data to be directed to the correct group.

The architecture of the database machine in Fig.2.7. shows a ring of processors S, B, I, and C. Each of the processors which form the ring will run a high priority process which undertakes the routing of information. Data is executed from the entities as a stream of tuples. Thus, the stream of tuples can be passed round a segment of the ring from the entity to the buffer processors. Therefore, a stream of a pipeline is formed which once it has been filled will cause a continuous stream of data to appear at the buffer process. Thus, the buffer process enables ring transmission fluctuations to be smoothed so that the operations processors J are always busy.

The entity processors E comprise a disk subsystem together with an M212 disk controller transputer. The M212 will also provide a small local cache for the disc and the related selection processor S may also provide a larger cache. A particular entity may be stored on a single disc subsystem or it may be partitioned across several such processors E. Such a partitioned entity can then have each of its partitions processed in parallel provided sufficient operations processors J are available.

B. Conclusion of Parallel Query Operation :

The architecture of parallel query operation enables the entities of a database to be stored on separate processors and further the entities can be partitioned onto several processors, thereby enabling parallel execution of the relational operators.

The architecture is reconfigurable in that a pool of processors is available to undertake the relational operations and these can be dynamically allocated as user queries are processed. Thus, a processor can undertake the operations; join, project and selection as required.

2.5. General Conclusion :

Since this research relates to two major research areas: object oriented databases and parallel processing techniques, it has only been possible to investigate a small number of existing systems which cover these areas, but these have been selected to illustrate the variety of research which is taking place.

Two sequential OODBMS have been discussed: EXODUS [Carey 86], and IRIS [Fishman 87] prototype DBMS. IRIS provides a number of different interfaces for users, including object SQL (OSQL). EXODUS, on the other hand, is designed as an extensible system which can be adapted to suit different applications.

Although the following systems: PRESTO [Bershad 88], Rekursiv [Harland 88], and CHANCER [Chalmers 89] depend upon object oriented paradigm, each implements parallelism in a different manner: PRESTO is designed for building distributed object oriented systems in a multiprocessor environment. But, Rekursiv implements parallelism through hardware, since it is a single board. On the other hand, the Chancer system implements parallelism by supporting the development of programs made up of a number of concurrently executing objects on a Meiko Computing Surface using Occam programming.

An application of parallel query processing on a Meiko Computing Surface [Kerridge 87] is described too.

CHAPTER 3

Parallel Processing Aspects In An Object Oriented Database Management System

3.1. Introduction :

As mentioned before, the implementation of an OODBMS in purely sequential systems inevitably leads to large problems involving efficiency. So, the aim of this chapter is to explore the different aspects of parallel processing that can be implemented within the scope of the OODBMS to gain high performance and efficiency.

In the following subsection, a general description of the PKBZ architecture will be discussed. Then, the different aspects of parallel processing will be illustrated, followed by the allocation and deallocation of the object classes in the main memory storage.

3.2. General Description of the PKBZ Architecture :

The PKBZ modules exist at three different levels. These levels are :

- a. External Management System (EMS)
- b. Object Management System (OMS)
- c. Storage Management System (SMS)

Fig.3.1 shows the block diagram of PKBZ modules.

3.2.1. External Management System (EMS) :

The EMS acts as an interface between the end user and the OMS. The end user can be any ordinary user like a cashier in a bank or an external system which is built on top of the PKBZ. Moreover, EMS allows the end user to retrieve, update and add data in a user friendly form. The EMS main functions are :

1. If the end user is not familiar with PKBZ messages (like a cashier at a bank), then the EMS acts as Query processor that transfers the external user request into a message which can be recognized by PKBZ. This transfer is performed through menu

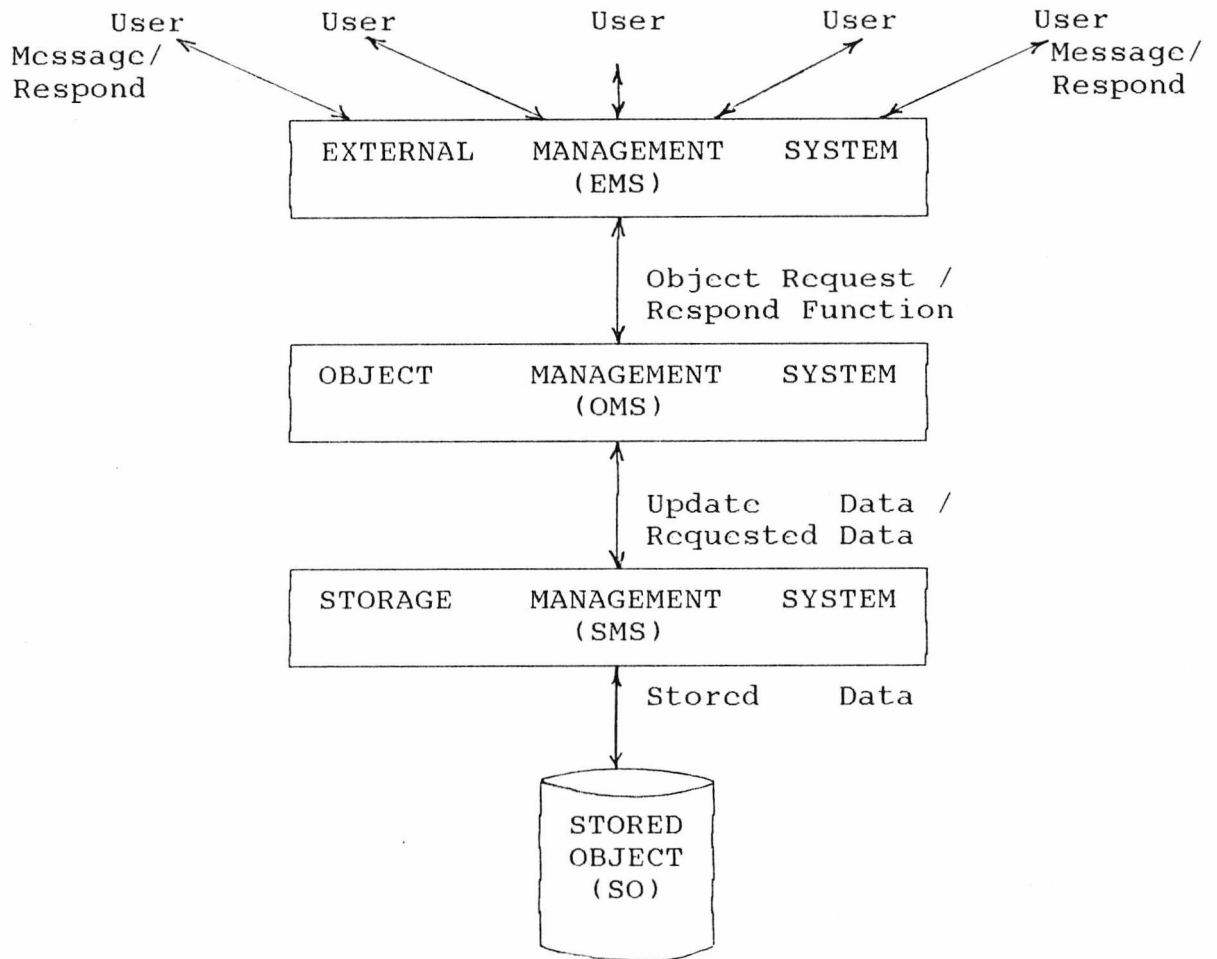


Fig.3.1 Generalised PKBZ Database System Architecture

driven screens. Each screen provides the end user with a set of tools to interpret his or her request. Then, after the dialogue is finished, the EMS interprets the user request into a message that can be understood by the PKBZ. The interpreted message consists mainly of the following parameters :

- a. The function or a method that the end user needs
- b. The object class ¹ in which the end user requests the message to be performed.
- c. Some parameters (or null) that are required according to the message type.

2. If the external user is another system which is built at the top of the PKBZ, then EMS can be regarded as a translator between the external user and the PKBZ. That is, every request is transformed into a message that can be recognized by PKBZ. Moreover, any further response from PKBZ is translated to the form that can be understood by the end user. Then, the message is stacked until it will be processed.

3. After the message is stacked and the OMS is ready, the message is sent to the OMS module for processing in the form of Object Request Function.

4. When the answer is received from OMS modules, the EMS will respond to the related end user.

5. EMS keeps track of every message until it is processed, or an error message is received.

3.2.2. Object Management System (OMS) :

The OMS controls the overall execution of Memory Objects ² in the PKBZ. This will involve, for example, Memory Objects allocation in the main memory storage as well as Memory Objects deallocation from the main memory. Such algorithms of allocation and deallocation will be discussed later. Moreover, when a user message is received from the EMS, the OMS directs the message to the corresponding Memory Object in the main memory.

¹ The object class can be either a system object class, or a user defined object class.

² The Memory Object is a unit of main memory storage in which an object class is stored. The schema, instances and methods associated with an object class are all encapsulated in a Memory Object.

Then, any further response from the Memory Object to the end user will be sent to the EMS.

Memory Objects, during run time, are grouped together in the form of an Object Group, since many applications require the ability to manipulate a set of object classes as a single logical entity for purpose of efficient storage and retrieval [Lorie 83][Kim 87]. The Memory Object grouping criteria will be discussed in detail later. On the other hand, if any Memory Object communicates with any other Memory Object(s), such communication will be controlled by OMS module.

3.2.3. Storage Management System (SMS) :

SMS can be regarded as a buffer between the logical representation of the data, in the form of Memory Objects in the OMS module and the physical data representation in Database secondary storage files. Moreover, SMS converts the physical data from the secondary storage to the Memory Object representation and vice versa.

The File Simulation Process in the SMS is responsible for fast accessing, retrieval, and updating data, since data is transferred from the secondary storage to the main memory in SMS, in the form of Pages or Page Objects. The File Simulation Process will be discussed in detailed later.

When OMS needs any data from the secondary storage, the OMS sends a system message to the SMS module, then the File Simulation Process retrieves data either from the main memory in SMS module, if it already exists, or from the secondary storage to the SMS main memory, then to the OMS module, if it does not exist in the main storage in SMS module.

The Stored Object consists of one or more Database files in the secondary storage in which data is stored in the physical format.

3.3. Different Aspects of Parallelism in PKBZ:

In this section, the different aspects of parallelism in the PKBZ will be discussed. The internal classifications of each aspect are illustrated too. Moreover, an example with each aspect will depict the idea of parallelism and the classification.

The Different Aspects of Parallelism in PKBZ can be classified in general into main five types. These types are :

- a. Parallel on Transputers
- b. Parallel on Single Transputer
- c. Parallel on Instructions
- d. Parallel on Data Distribution
- e. Parallel on Background

The following subsections describe the differences among the various types. The PKBZ architecture which is described in the previous subsection is used to demonstrate the different aspects of parallelism. These aspects can be adapted, in general, to any other OODBMS with different architecture as well as any software system, since most current software systems are built in the modular form.

3.3.1. Parallel on Transputers :

The Parallel on Transputers depends upon the distribution of system architecture upon various processors (transputers). For example, the architecture of PKBZ consists mainly of three different modules : EMS module, OMS module, and SMS module. Each module consists of sub-modules (processes). These processes can be grouped into different groups, say G_0 , G_1 , ... , G_n . Hence, one or more groups of processes is mapped onto a single processor (transputer).

The Parallel on Transputers does not depend upon how these groups are executing during the run time. But Parallel on Transputers depends upon the distribution of the system architecture.

The Parallel on Transputers type can be classified according to the system architecture, grouping into :

- a. Homogenous Parallel System Architecture
- b. Hybrid Parallel System Architecture

The following subsections describe the difference between the two types of the system architecture grouping.

A. Homogenous Parallel System Architecture :

In Homogenous Parallel System Architecture, all processes in each group belong to only one module of the three PKBZ modules. The Homogenous Parallel System Architecture is classified according to the transputer allocation into :

- * Simple Homogenous Parallel System Architecture
- * Complex Homogenous Parallel System Architecture

The following subsections describe the two different allocation techniques.

** Simple Homogenous Parallel System Architecture:*

In Simple Homogenous Parallel System Architecture, each PKBZ module forms a single group. Then, each group is allocated to a single transputer. Fig.3.2³ depicts the idea. That is, all the processes of the EMS module, OMS module, and SMS module are allocated to three transputers T1 , T2 , T3 respectively.

The Messages Module in the EMS analyses each user's message and then sends the message to the OMS. The Messages Module consists internally of different sub-modules. Such organisation of Messages Module depends upon the various message functions.

In the OMS, each Object Group contains a set of Memory Objects. The Memory Object grouping depends upon different difficult criteria. Such grouping criteria will be discussed later.

Moreover, the data is stored in SMS into groups too. This grouping will be described later. The SMS is linked to the Stored Objects in the secondary storage database file.

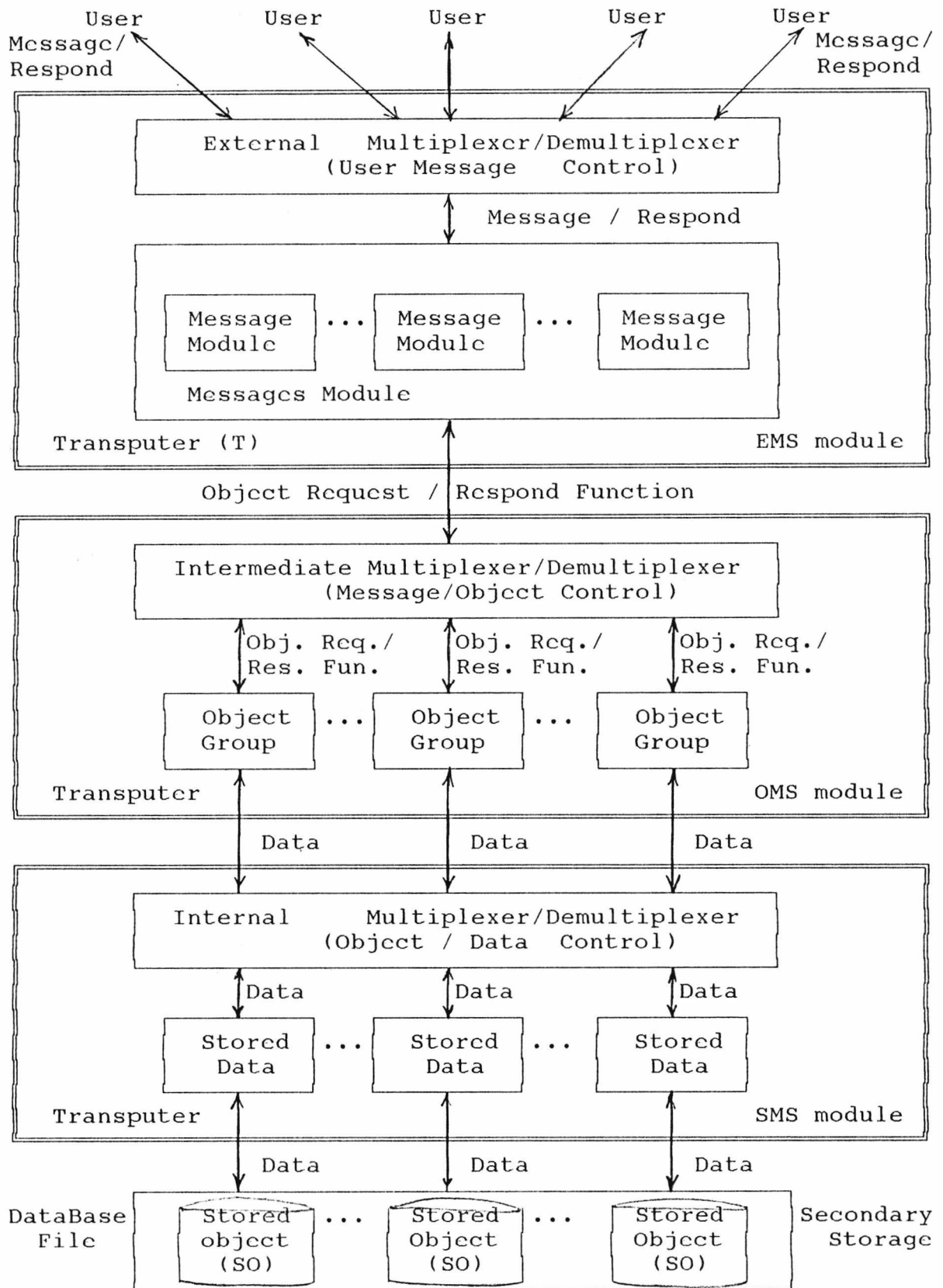
The execution of any user's message will be performed as follows :

1. One or more Users (U) send the message to the EMS

2. The External Multiplexer passes the message to the Messages Module. The

External Multiplexer acts as a controller between the external users and the other processes in the EMS.

³ The Transputer has only four external pair channels. But, in the figure, the number of channels per transputer is greater than four since the diagram has been simplified



N.B. There is no 1-1 mapping between blocks in different modules

Fig.3.2 Simple Homogenous Parallel System Architecture

3. The Messages Module checks the syntax of each message and carries out the necessary processes corresponding to each message. Then, the Messages Module sends the message to the OMS.
4. The OMS analyses the message and translates the message into an Object Request Function. Then, the Object Request Function is sent to the relevant Memory Object in the Object Group.
5. The relevant Memory Object receives the Object Request Function and invokes the corresponding method according to the function request.
6. If any Memory Object needs any data from another Memory Object, a request message will be "mailed" through the Intermediate Multiplexer/Demultiplexer, since Intermediate Multiplexer/Demultiplexer not only operates as a controller of communication between the EMS and the OMS, but also among Memory Objects.
7. When a Memory Object completes the Object Request Function, the result will be sent to the Intermediate Multiplexer/Demultiplexer and then to the Message Module which responds to the end user.
8. If any Memory Object needs data from the Stored Data in the SMS, the Internal Multiplexer/Demultiplexer acts as a manager between the Memory Objects and the Stored Data in SMS.
9. The SMS stores and distributes the data among Stored Data processes.

The Simple Homogenous Parallel System Architecture has both an advantage and a disadvantage.

** Advantage :*

1. The Simple Homogenous Parallel System Architecture is a simple method of exploring parallelism either with a PKBZ, or any other software system, since each module of the PKBZ is mapped directly to a single transputer.

** Disadvantage :*

1. Only three different processors (transputers) are used. However, the PKBZ can

be mapped onto more than three processors to gain high performance of the parallel processing. The Complex Homogenous Parallel System Architecture depends upon this concept.

* *Complex Homogenous Parallel System Architecture :*

In Complex Homogenous Parallel System Architecture, each Homogenous Parallel System Architecture group is allocated to one or more transputers. Such distribution is expected to be better than Simple Homogenous Parallel System Architecture, since the processing of each module is distributed among different processors. It is clear that there are many Complex Homogenous Parallel System Architecture distributions, since there are many different object grouping combinations. Hence, Fig.3.3 illustrates one of such distributions.

In Fig.3.3, the EMS module is allocated to a single transputer. But, the OMS module is mapped onto more than one transputer, since the organization of the OMS depends mainly upon the Object Group and each Object Group is processed in a single transputer. Similarly, the SMS module is mapped onto more than one transputer too, the dispersion of the SMS depends mainly upon Stored Data.

It is clear from the Simple Homogenous Parallel System Architecture diagram (Fig.3.2) and Complex Homogenous Parallel System Architecture (Fig.3.3) that the main difference between the two techniques is the number of transputers in each system. But, the processes are similar. So, the user's message will be performed as described before in Simple Homogenous Parallel System Architecture.

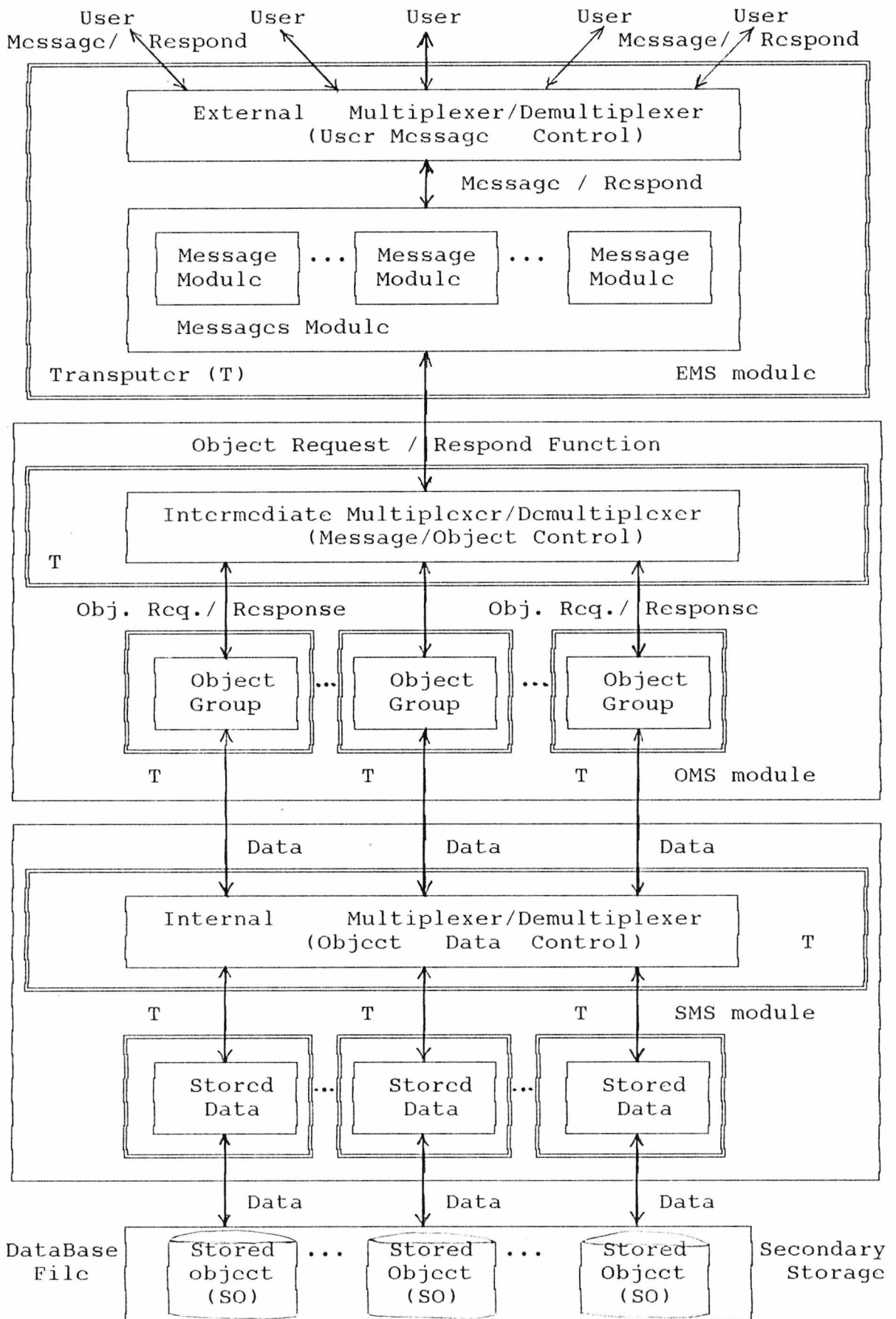
In Complex Homogenous Parallel System Architecture, the Object Group can be either :

i. Static Group

or ii. Dynamic Group (Migrate Object)

* *Static Group :*

The system structure will be as illustrated in Complex Homogenous Parallel System Architecture. But, once an Object Group is loaded in the system, the Object Group remains



N.B. There is no 1-1 mapping between blocks in different modules

Fig.3.3. Complex Homogenous Parallel System Architecture

in the same memory location. That is, the Object Group is static in the main memory. It is clear that this is a simple technique.

* *Dynamic Group :*

In Dynamic Group, the Object Group or a subset of Object Group can migrate from one transputer to another transputer. This migration can be performed, for example, in the following cases :

- i. To balance the load across all transputers with regard to the number of Memory Objects in each transputer.
- ii. To balance the processing among the transputers, if one transputer is inactive and another has a heavy processing load at the same time.

In order to implement Dynamic Group, an intelligent process has to monitor both the processing and the Memory Object load across the transputers during the execution time.

The Complex Homogenous Parallel System Architecture has both an advantage and a disadvantage which can be summarized in the following :

* *Advantage :*

1. The processing is distributed among different transputers. Therefore, the efficiency is expected to be better than Simple Homogenous Parallel System Architecture.

* *Disadvantage :*

1. In an implementation, an extra control has to be implemented to manage the flow of the data between different transputers. These controls can be regarded as an overhead upon the system implementation.

B. Hybrid Parallel System Architecture :

In Hybrid Parallel System Architecture, all the processes in each group do not necessarily belong to only one module of the three PKBZ modules. For example, some

processes in OMS module and another in SMS module can be mapped onto the same group. It has to be noted that there must be some association between these processes, otherwise such grouping will be nonsense.

For example, it is clear that Memory Objects are grouped in OMS module and each Memory Object needs data from the corresponding stored data in SMS. Thus, in order to reduce the communication time between any Object Group and the related Stored Data both of them can be grouped into the same transputer. The Fig.3.4⁴ depicts the idea.

The execution of any user's message will be performed as follows :

Steps from 1:7 will be the same as in Simple Homogenous Parallel System Architecture

8. If any Memory Object needs data from the corresponding Stored Data, it communicates directly through software channels, since both Memory Object and Stored Data are gathered in the same transputer.
9. The Stored Data communicates with the Stored Objects in the external database file through Internal Multiplexer/Demultiplexer (Data Stored Control)

In Hybrid Parallel System Architecture, the Static Group and Dynamic Group technique can be applied too. But, in Dynamic Group, both Object Group and Stored Data are migrated together.

The main advantage of Hybrid Parallel System Architecture will be :

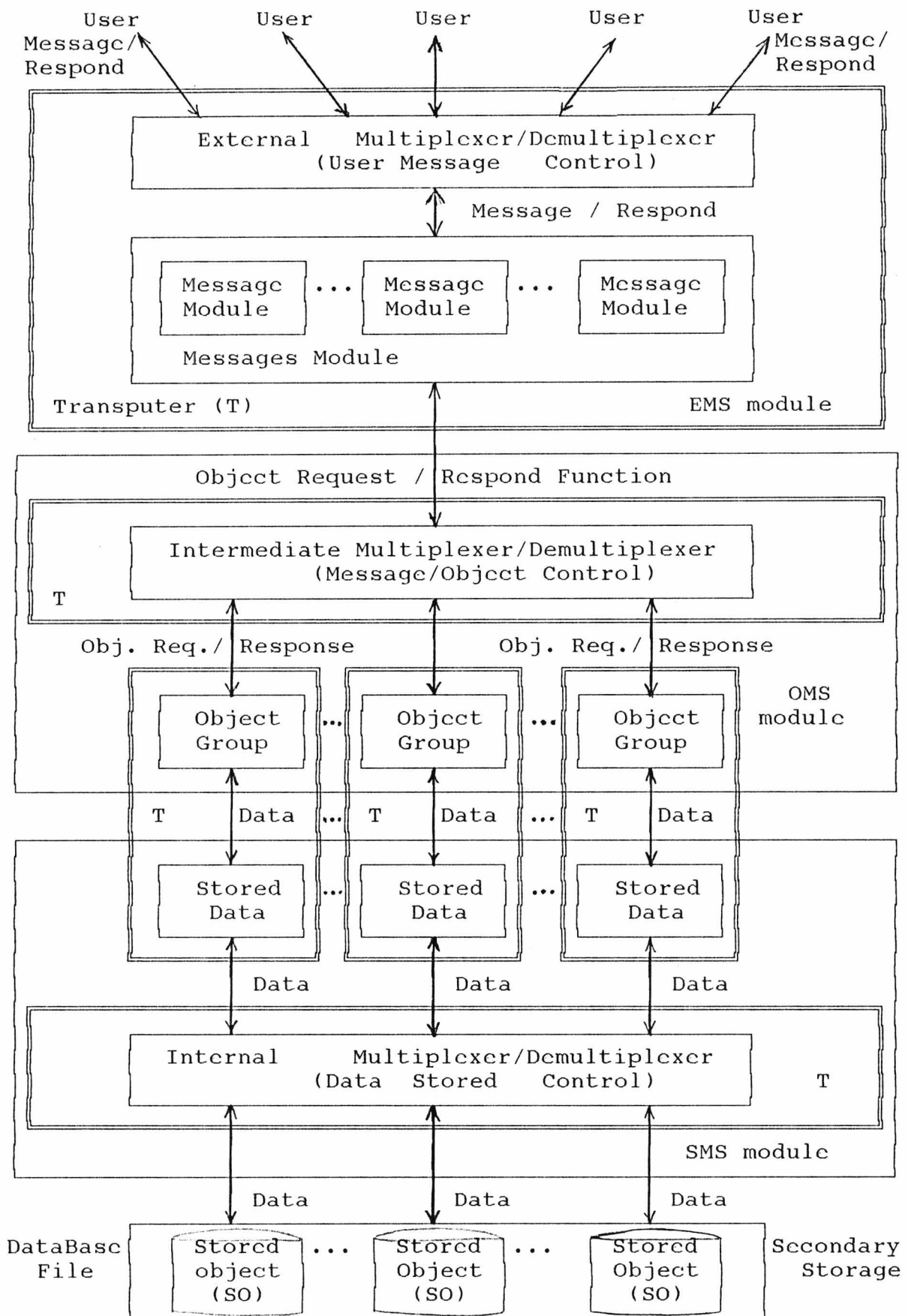
* *Advantage :*

1. The distribution of the processes is not limited to the number of the PKBZ modules as in Simple Homogenous Parallel System Architecture. So, it is more flexible than Simple Homogenous Parallel System Architecture.

3.3.2.Parallel on a Single Transputer:

In Parallel on Transputers, the parallelism depends upon mapping the system

⁴ See Footnote 3



N.B. There is no 1-1 mapping between blocks in different modules
Fig.3.4 Hybrid Parallel System Architecture

architecture onto different transputers, so the parallelism occurs when more than one transputer are executed in parallel. However, there is another degree of parallel processing within each transputer itself, since each transputer can carry out parallel activities. For example, the transputer can execute normal instructions, input on four links and output on another four links simultaneously [INMOS 84].

Furthermore, some processes of PKBZ are nearly independent of each other. So, there are different processes that can be running in parallel within each transputer. These processes communicate internally within a single transputer via software channels, whereas the communication with the other transputers is performed through hardware channels. The software channels are implemented by allocating bytes in the main memory on each transputer, while the hardware channels are implemented by hard channels between transputers. [Burns 88] describes the two different types of software and hardware channels in details.

So, in Parallel on Single Transputer type some processes can input from either software channels or hardware channels, while other processes can output to other processes. Simultaneously, other processes can be running in parallel. The integration of such processes, within a single transputer, gains a high performance.

This type of parallelism depends upon the hardware (transputer) as well as the software (mapping the processes onto a single transputer). Fig.3.5 illustrates an example of the Parallel on Single Transputer type in which each module communicates with a different external user in parallel. So, each user can perform a different job at the same time.

The following code in the Occam language [Jones 87] depicts this type of parallelism⁵ :

```
PAR      -- Parallel Processing follows
  PAR i = 0 FOR 3 -- Three Parallel EMS Process
    EMS(screen[i], keyboard[i], ... , ... )
  Mult.Demu(... , ... ,           , ... )
```

All the processes are mapped onto a single transputer.

⁵ The indentation is more than two spaces to clarify the Occam code

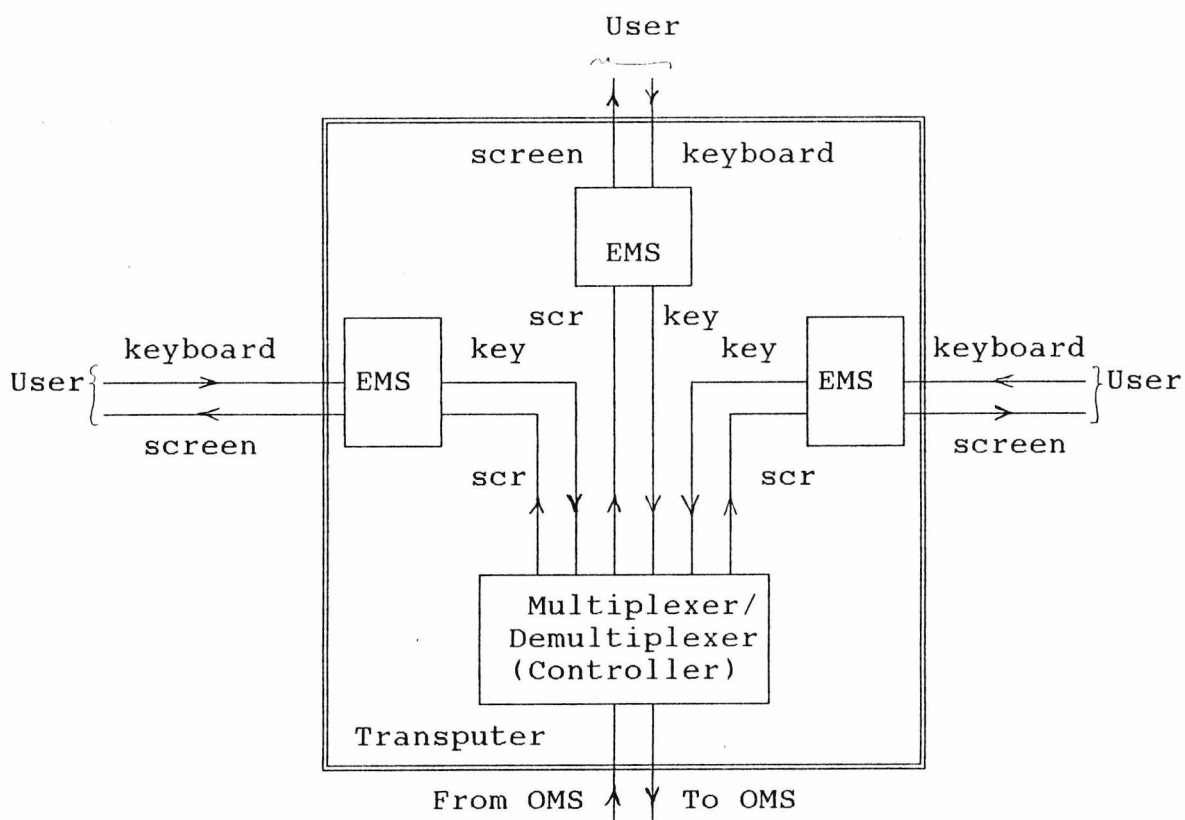


Fig.3.5 Parallel on Single Transputer

3.3.3. Parallel on The Instructions:

In this type of parallelism, the parallel processing depends upon the instructions. In some situations, the processing within a single process may be completely independent. So, it is better to perform such processing instructions in parallel. For example, in "AirLine Transportation Example", CREW-NAME inherits CREW and NAME (which are referred to as the "inherit" object classes of CREW-NAME), while CREW-NAME is inherited by STAFF (which is referred to as the "inherited by" object class of CREW-NAME). All object classes include references to both the "inherit" object classes and the "inherited by" object classes. These "inherit" object classes data and "inherited by" object classes data can be initialized in parallel before a new object class is created. The two initializations are independent of each other. The following code in Occam language depicts the Parallel on the Instructions type :

```
PAR                                     -- Parallel Processing follows
  PAR i = 0 FOR (SIZE inherit)         -- Initialize inherit
    inherit[i] := 0
  PAR j = 0 FOR (SIZE inherited.by)    -- Initialize inherited by
    inherited.by[j] := 0
```

It is clear that both the arrays must be declared elsewhere in the process before the initialization processing is carried out.

Furthermore, not only are both the "inherit" array and "inherited.by" array initialized in parallel, but also each element in both arrays is initialized in parallel too.

However, this type of parallelism is called Weak Parallel process, since the related instructions are executed entirely within a single process within a single transputer. Moreover this type of parallelism depends upon software. That is, this feature depends upon how much parallelism the implemented language physically supports.

* *Disadvantages :*

1. This type of parallel processing depends mainly upon the implemented language.
2. The processing is carried out within a single transputer within a single process.

3.3.4. Parallel Data Distribution:

Parallel processing on the transputers not only provides huge processing power, but also supplies the system with a large volume of main memory storage capabilities, since each transputer main memory may be from 1MB to 4MB at present, using sixteen transputers will provide a system with between 16MB and 64MB of main storage. Such a high storage of main memory can be used to get a large volume of data in the main memory storage as well as processing these data in parallel. Hence, this facility is used to load a large number of Memory Objects in the main memory. Then, any further operation can be performed directly in the main memory exploiting parallel processing where possible. This type of parallel processing is called Parallel Data Distribution.

The following example illustrates Parallel Data Distribution. Let us suppose that in "AirLine Transportation Example" the distribution of the structural objects related to an entity aggregate object STAFF among transputers will be as shown in Fig.3.6 and a new staff member has to be added in the database. When an "ADD.INSTANCE" message is sent to the STAFF object with the new staff member data, the STAFF object will ask the PKBZ to generate a new surrogate corresponding to the new staff, since the surrogate is used as a link (key) among the related structural objects instances in STAFF object. Then the following code can be used to add new instances to the related structural objects :

```
PAR      -- Parallel Processing follows
a ! ADD.INSTANCE ; EXPERIENCE ; surrogate ; type
SEQ
  b ! ADD.INSTANCE ; CREW-NAME ; surrogate ; name
  b ! ADD.INSTANCE ; LOCATED-AT ; surrogate ; address
c ! ADD.INSTANCE ; CREW-RANK ; surrogate ; rank
d ! MESSAGE
```

The instance data related to each Memory Object is sent in parallel through hardware channels between transputers. The "ADD.INSTANCE" is a tag protocol to identify the message, while the object name is the second parameter and the instance data is the last two parameters. So, all instances are added in different Memory Objects simultaneously, since all processes at that time are independent of each other. It has

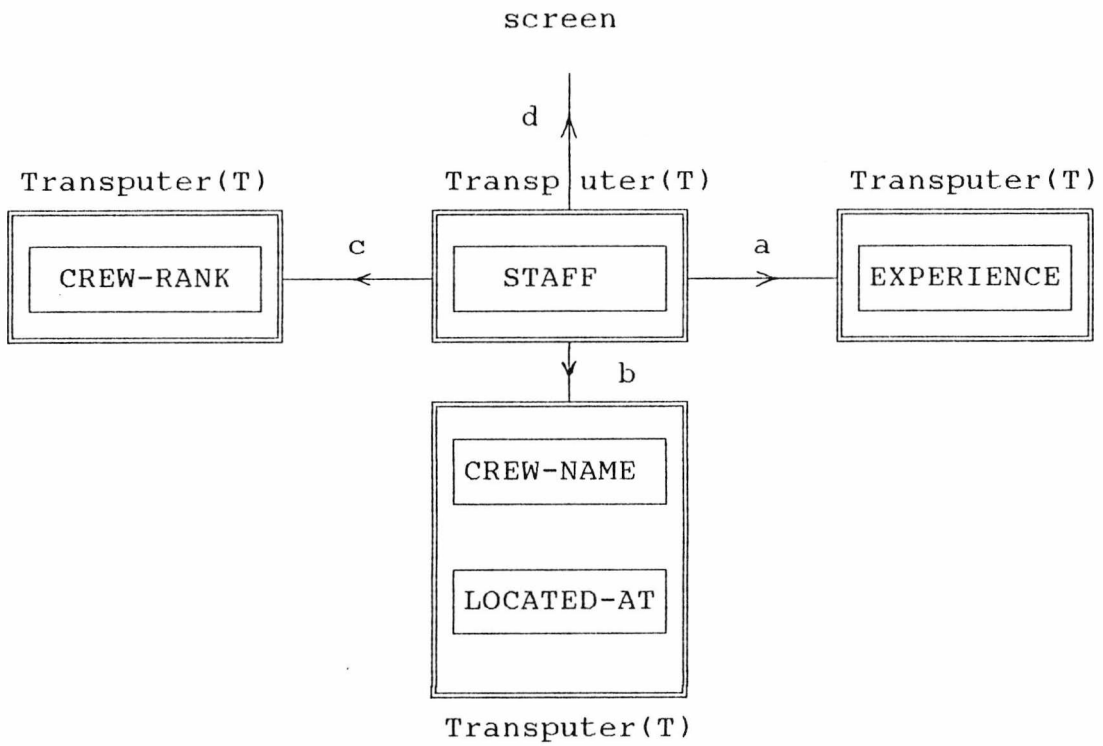


Fig.3.6 Distribution of the structural Objects Related to STAFF Object among Transputers

to be mentioned that although the instances related to the CREW-NAME and LOCATED-AT Memory Objects are sent in sequence, the addition instance process, in both Memory Objects, is carried out in parallel using Parallel on a Single Transputer type as described before, since both the processes are independent of each other.

It is clear that the variable MESSAGE must be defined as a string which carries a response such as "A new staff instance is added". The MESSAGE can be sent to the screen either in sequence to ensure that the adding processes have executed properly, or in parallel as shown above. But, if MESSAGE is sent in parallel, a new type of parallel processing called Parallel on Background type will be used. This type of parallelism will be discussed in the next subsection.

** Advantages :*

1. The full power of parallelism is achieved, since independent processes are distributed among various transputers (processors).

2. In the some sequential machines, the DBMS swaps the data between the secondary storage and the main storage memory, since the main memory cannot accommodate such large volume of data (e.g. 64MB). Using Parallel Data Distribution, the data swapping frequency is expected to be less than in these sequential processing.

3.3.5. Parallel on Background :

In any traditional sequential database system, the user's request is translated into a sequence of procedures. Each procedure is performed one after the other. When all procedures are completed successfully, the end user receives an acknowledgement. But if any error occurs during the processing, an error status is sent to the end user. Since a single processor performs all the tasks, the response time of any user's request is the summation of all the times needed to perform each procedure. But in parallel processing the response time can be improved not only by processing the user's request in different processors simultaneously, as shown in Parallel Data Distribution, but also by executing some processes in the background during the system life time. So, the average response

time of the user's request is expected to be less than in traditional sequential database system.

In Parallel on Background, the external user is not aware of the processes that are carried out in the background during the life time of the system execution. It has to be noted that the parallel background processes must be selected carefully and must be secure, otherwise the integrity of the database may be corrupted.

The following are the possible processing mechanisms for Parallel on Background:

- a. Complete/Incomplete Parallel Processing
- b. Object Update Parallel Processing
- c. File Simulation Processing

The Complete/Incomplete Parallel Processing is performed in all PKBZ modules whenever it is possible. On the other hand, the Object Update Parallel Processing is carried out in the OMS module, while the File Simulation Processing is executed in the SMS module. The following subsections explain the difference among these techniques.

A. Complete/Incomplete Parallel Processing:

Any user's message has two different views. These views are the user's view and the system view. So, at a certain time during execution of a specific message, the message is completely executed from the user's view point. However, from the system view, it may be incomplete. To explain such a strange characteristic, let us consider that the user's message requests to open a database. In order to open a database the message is interpreted, in general, as the following internal procedures:

- a. Check the message parameters
- b. Check the database existence in the secondary storage
- c. Read the system object ⁶ classes from the secondary storage
- d. Create new Stored Data in SMS module to load system object classes data
- e. Change the data from physical data format into a Memory Object format
- f. Create new Memory Objects in OMS corresponding to the system object classes

⁶ System Object classes store information about each user's object class

g. Respond to the end user by message stating that the database is opened

In the traditional sequential database, the response time is the summation of all the times taken to perform each one of the previous procedures. But in Parallel on Background type, once the first two procedures are carried out and the database is known to exist, the PKBZ could perform step "g" directly, and the end user is informed that the message is complete, since it is complete from user's view. But from the system view the message is not complete. So, the steps c, d, e, and f are carried out in the system background. It is clear that the response time is reduced by using Complete/Incomplete Parallel Processing.

** Advantage :*

1. Reducing the average time taken to process a user's message

** Disadvantage :*

1. In the unlikely event that an error occurs and the background processes are not performed successfully, then the user will have received an incorrect message

B. Object Update Parallel Processing:

One mechanism for Parallel on Background is the distribution of the Object Groups in the OMS module. As mentioned before, each Object Group consists of a collection of Memory Objects. Once a Memory Object is opened, it is loaded into the main memory. Then, all the processing is carried out in the main memory. It is clear that not all Memory Objects are executing at the same time. This characteristic attracts a new type of parallel background process. Hence, a new process called an Object Manager is executed in the background during the system life. As shown in Fig.3.7, there is only one Object Manager per Object Group, since the Object Manager checks periodically every Memory Object in the Object Group. So, if the instances are updated or some of the schema information is changed, these changes will be reflected to the SMS module and in consequence to the Stored Objects in the secondary storage database file. Fig.3.7

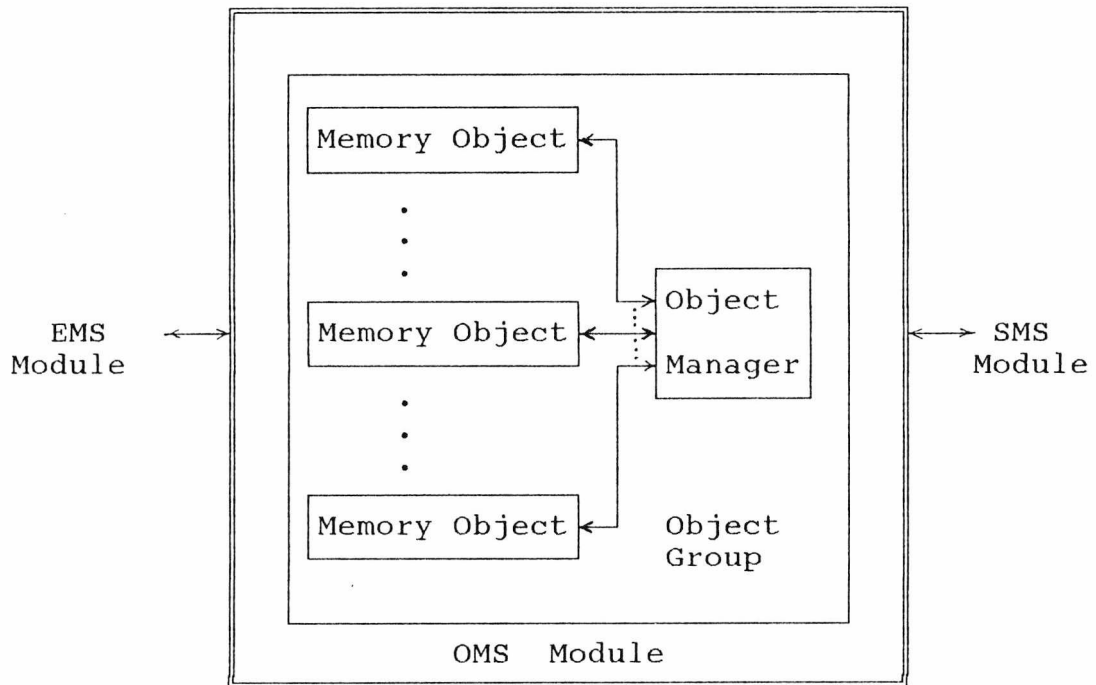


Fig.3.7 Object Update Parallel Processing
Block Diagram

illustrates Object Update Parallel Processing block diagram.

Generally speaking, out of all the Memory Objects brought into the memory, only those that have been modified will be saved back to the secondary storage, unlike the ZEITGEIST [Ford 88] OODBMS, where all Memory Objects brought into the memory are saved.

* *Advantages :*

1. Updating mechanism is carried out periodically. So, if there is more than one updating process in a single Memory Object, these processes are executed in a single step. Similarly in the ENCORE OODBMS [Hornick 87], the updating process is performed as a package to minimize the amount of network traffic and reduce the amount of processing execution.

2. Idle, or virtually idle Memory Objects in the main memory are processed for updating, while non-idle are left. So, all Memory Objects are executing in parallel from the system view point. In the OZ+ [Weiser 89] OODBMS, for example, the system gives each Memory Object an activity rating which characterizes the idle or virtually idle Memory Objects.

C. File Simulation Processing:

Since any database needs normally a large volume in the secondary storage, accessing the secondary storage slows the database performance. This characteristic can be improved by using File Simulation Processing. The database file in the secondary storage is divided into pages. Each page consists of records. Thus, the File Simulation Processing can be regarded as a process that is executed in parallel, in the system background, in the SMS module. When a new page is read or written to the secondary storage for the first time only, a new page is allocated in the main memory of SMS module. Hence, any further reading or updating to the same page is processed from the main memory. In consequence, the performance is expected to be improved. Fig.3.8 shows the block diagram which illustrates the File Simulation Processing.

The File Simulation Processing is a way of illustrating parallel processing in the system background, since any page can be read or updated in the SMS module, while

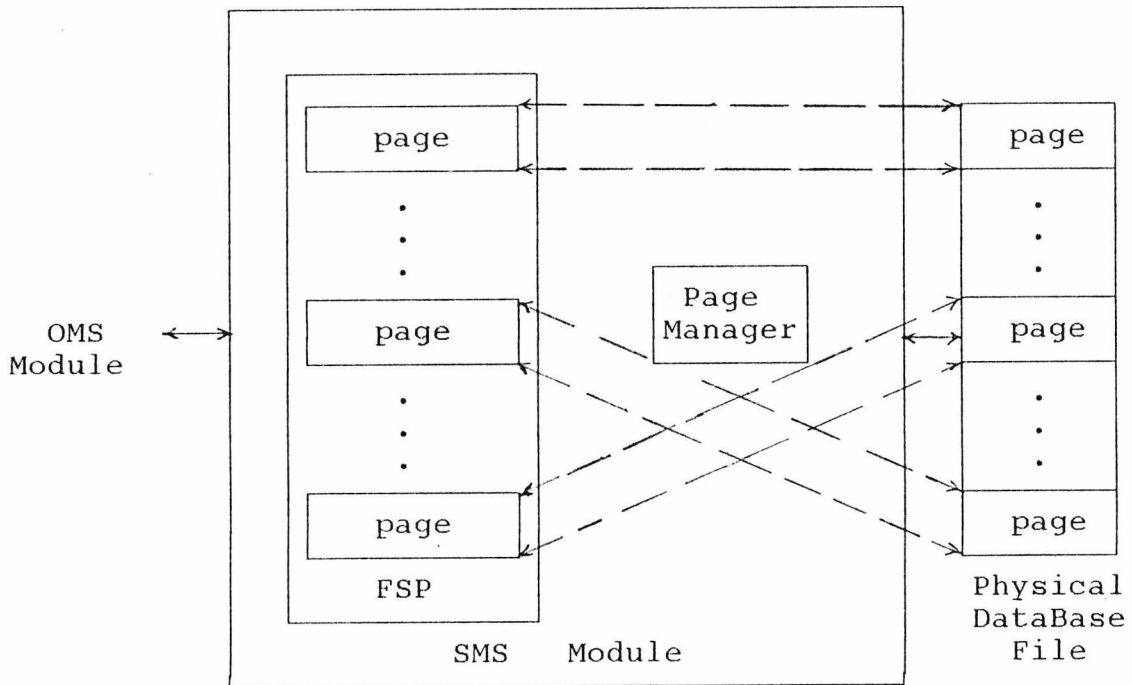


Fig.3.8 File Simulation Processing (FSP) in the SMS Module in the form of pages

another page is updated in the secondary storage. Moreover, File Simulation Processing can be considered as parallel input/output, since more than one page can either be read or written simultaneously in the SMS module.

Moreover, a process called a Page Manager is executing in parallel in SMS module. Its main function is nearly equivalent to the process Object Manager in the OMS module. But, the Page Manager monitors each page periodically and any update is reflected to the secondary storage.

Hence, the SMS module keeps track of the following information for each page in the main memory:

1. The number of used records on each page
2. The position inside the page to store new information
3. The location of the page in the secondary storage file
4. Some other information like a flag to indicate if the page is updated or not and so on.

Besides the previous information, some functions have to be implemented to perform the following operations within each page:

1. Read a record from a page
2. Write a new record to the page
3. Update existing record in the page
4. Delete a record or part of the record in the page

Moreover the following constraints must be fulfilled :

1. When adding a record, the number of records currently in the page must be less than or equal to the maximum number of records within each page
2. The size of each record must be less than or equal to the maximum size of the record in the physical file.

It is clear from the previous information that the implementation of File Simulation Processing will be a complicated process, since a huge amount of information for all pages has to be stored, the functions required for each page must be implemented,

and the constraints of each page have to be fulfilled. In order to simplify the problem of implementation, each page in the main memory storage will be regarded as a single object class (i.e. Memory Object) called a Page Object. Within each Page Object, the functions will be defined as system functions, the records will be implemented as instances of the Page Object, the constraints will be applied to the instances and any further information can be defined in the schema description within each Page Object. Fig.3.9 shows the modified File Simulation Processing in the form of Page Object.

The aim of the previous modification of File Simulation Processing is to build the SMS module in the form of Memory Objects too. So, both OMS module and SMS module are in the form of Memory Objects. It should be noted that not all object oriented systems are implemented in this way. For example, the current Storage Manager in Iris [Fishman 87] is a conventional relational storage subsystem.

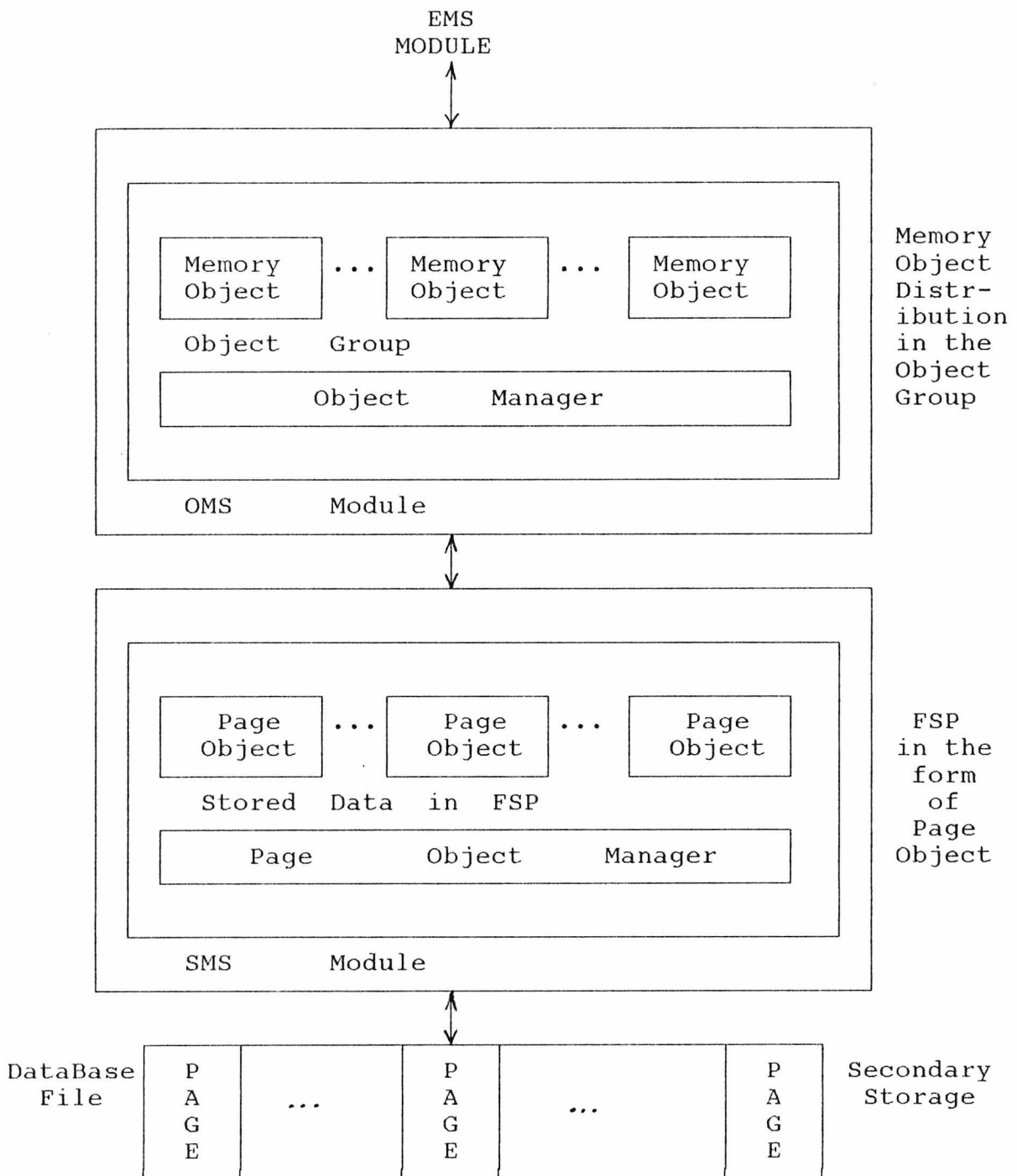
In File Simulation Processing, a process called Page Object Manager is executed in parallel in the background of the system. Its function is similar to the process Object Manager in the OMS module. That is, any update in a Page Object is monitored and the updating will be reflected to the physical Database File in the external storage.

It is clear from the previous discussion and Fig.3.9 that the internal block diagram of the Stored Data in SMS module will be the same as the internal block diagram of the Object Group in OMS module and that is one reason for the modification of the File Simulation Processing to be in the form of Page Objects so that both the Object Group process and Stored Data process may be similar with some minor changes.

The File Simulation Processing illustrates the file storage as a parallel process in Parallel on Background type. The Parallel on Background advantages are :

** Advantages :*

1. The response time for an external user is reduced
2. The process that depends upon the mechanical motion (update secondary storage) is performed in the background.



N.P. There is no 1-1 mapping between blocks in different modules

Fig.3.9 File Simulation Processing (FSP) in the form of Page Object

3.4. Memory Object Grouping in PKBZ :

As described in the previous subsections Memory Objects are grouped into Object Groups in OMS module and Stored Data in the SMS module. In this subsection, the following Memory Object criteria will be illustrated :

1. Type of Memory Object grouping
2. Allocation of Memory Object
3. Deallocation of Memory Object

The discussion of each criterion will be provided in the following subsections.

3.4.1. Type of Memory Object grouping :

Memory Objects are grouped in the main memory storage according to different criteria. The different criteria for Memory Object grouping are :

- a. Inheritance Structure Grouping
- b. Non-Inheritance Structure Grouping
- c. Random Grouping
- d. Sequential Grouping

In all the previous groupings the load has to be balanced. That is, the number of Memory Objects in each group should be nearly equal, if it is possible (balance condition). The following subsections describe the different criteria.

A. Inheritance Structure Grouping :

Many applications require the ability to define and manipulate a set of Memory Objects as a single logical entity for the purpose of efficient storage and retrieval [Kim 89], [Lorie 83], [IEEE 85], [Kim 87]. [Hornick 87] explains how Memory Objects are grouped in segments in ENCORE OODBMS.

So, in the Inheritance Structure Grouping, the Memory Object in the same inheritance structure is grouped into the same object grouping.

For example in "AirLine Transportation Example", all object classes in the inheritance path of the STAFF object are grouped in the same Object Group. Similarly,

all object classes in the inheritance path of AIRBUS, TRIP, and SCHEDULE objects are grouped in different Object Groups. The common object classes such as CREW, PLANE, ATRIP, ROUTE, and TYPE are grouped in one Object Group only.

** Advantage :*

1. Memory Objects in the same inheritance path need normally to communicate with each other. So, the communication is carried out through software channels, if it is possible. That is, all the inheritance path objects are near to each other.

** Disadvantage :*

1. The criteria depends upon the inheritance structure of the Memory Object, so each Memory Object's inheritance has to be checked first before loading in the main memory.

2. When the search for many instances in different Memory Objects is executed, the processing will be performed in a single transputer only, since as described before in Parallel System Architecture an Object Group is mapped onto a single transputer.

B. Non-Inheritance Structure Grouping :

In the Non-Inheritance Structure Grouping the Memory Objects in the object group do not belong to the same inheritance structure, if it is possible. For example, let us consider "AirLine Transportation Example", the Memory Objects in the inheritance structure of STAFF object are distributed among the different Object Groups. Similarly in the AIRBUS, TRIP, and SCHEDULE objects, the Memory Objects in their inheritance structure are distributed in different Object Groups.

** Advantage :*

1. In Complex Homogenous Parallel System Architecture and Hybrid Parallel System Architecture, the object groups are distributed among different transputers. Then, the processing is distributed too, when any process is performed within the same inheritance structure.

* *Disadvantage :*

1. The criteria depends upon the inheritance structure of the Memory Object, so each Memory Object inheritance has to be checked first before loading to the main storage.

C. Random Grouping :

The distribution of Memory Objects is random. There is no constraint in the distribution. The only constraint is the balance condition. This type of criterion suits grouping of the Stored Data in File Simulation Processing, since there is no inheritance structure between Page Objects.

* *Advantage :*

1. It is a simple criteria.

* *Disadvantage :*

1. Since, there is no criteria in Memory Object grouping, it may happen that some Object Groups have heavy processing, while others are idle.

D. Sequential Grouping :

The Sequential Grouping is suitable only for grouping in Stored Data in File Simulation Processing. In Sequential Grouping, the Page Objects which have pages in the secondary storage in sequence are grouped together, if it is possible. That is, the first n-pages in the secondary storage are mapped onto different Page Objects, and these Page Objects are grouped together in the same group, and so on.

* *Advantage :*

1. It is a simple criteria

2. The updated Page Objects are reflected in the secondary storage in sequence.

Then, the head movement in the disk is expected to be optimum.

3.4.2. Allocation of Memory Object :

The allocation of Memory Object depends upon the following parameters:

- a. The internal memory of each transputer
- b. The number of transputers allocated to the OMS module
and the SMS module.
- c. When Memory Objects are allocated in the main memory storage

The first parameter is defined by the system hardware. But the second parameter is defined by the OODBMS designer. In the third parameter there are different mechanisms for loading the Memory Objects in the system as follows :

- a. Demanded Object
- b. Inherit Structure Object

A. Demanded Object :

In Demanded Object, the Memory Object is allocated into the memory, when the user or the system demands the corresponding object class. Then, the Memory Object is grouped into the related group according to the criteria described in the previous subsection.

For example, in "AirLine Transportation Example", the STAFF object is allocated only, if the user sends a message to the STAFF object.

** Advantage :*

1. It is a simple technique.
2. Only demanded object classes are allocated.

B. Inherit Structure Object :

Normally, an object class is not executing alone. An object class interacts with other object classes in the same inheritance structure. So, when an object class is demanded, not only the demanded object class is allocated, but also the inherited object classes are allocated too.

* *Advantage :*

1. When a user's message requests information from other object classes in the same inheritance structure, the inherited object classes are already allocated, since all inherited object classes are normally executing together.

* *Disadvantage :*

1. Some allocated Memory Objects may not be required during system execution.
2. The inheritance path must be checked when a new Memory Object is allocated in the system.

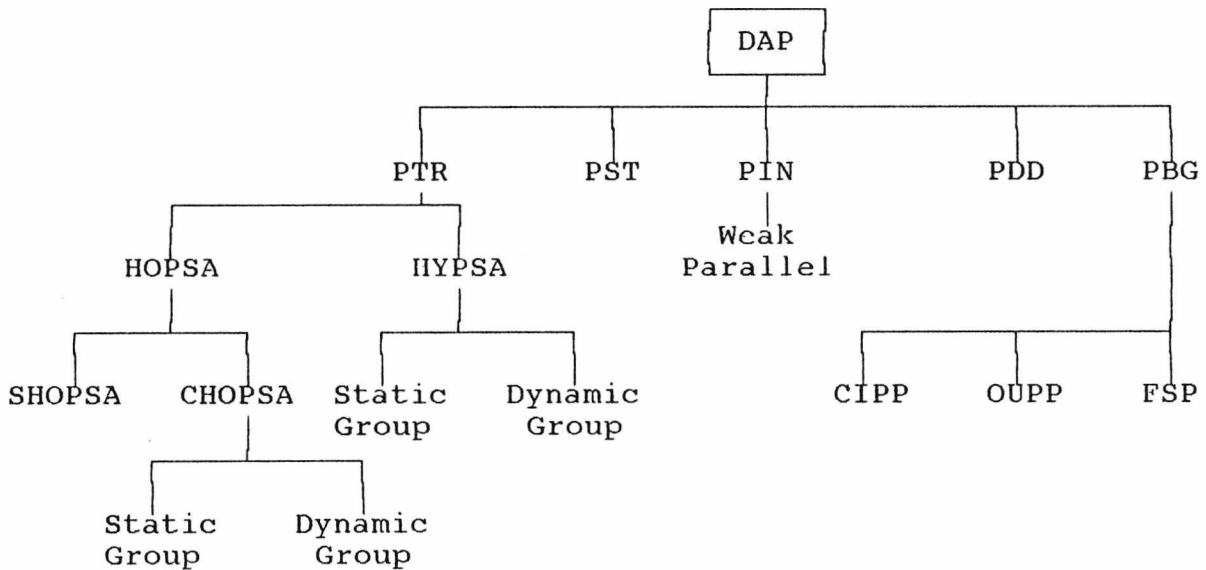
3.4.3. Deallocation of Memory Object :

If the number of Memory Object required is greater than the maximum number of the Memory Object in the system, some Memory Object will be deallocated and the deallocated Memory Objects will be restored back into the secondary storage file. The different criteria for choosing the required deallocated Memory Object has an analogy with page deallocation in the operating system. So, the same technique in [Janson 85] can be applicable too. For example, Last Recently Used (LRU), First In First Out (FIFO), and so on. These criteria are beyond the scope of this research and can be investigated in future research.

As an example, though, in the OZ+ [Weiser 89] OODBMS, the system gives each Memory Object an activity rating, which characterizes the frequency with which a Memory Object is accessed. Memory Objects that are virtually inactive are deleted from the memory after updating the corresponding data in the secondary storage.

3.5. Conclusion :

This chapter has explored the different aspects of parallelism that can be implemented in PKBZ. Fig.3.10 summarizes the various aspects of parallelism in the graph form.



- DAP . . . Different Aspects of Parallelism
- PTR . . . Parallel on TRansputers
- PST . . . Parallel on Single Transputer
- PIN . . . Parallel on the INstructions
- PDD . . . Parallel Data Distribution
- PBG . . . Parallel on BackGround
- HOPSA . . . HOMogenous Parallel System Architecture
- HYPSA . . . HYbrid Parallel System Architecture
- SHOPSA . . . Simple HOMogenous Parallel System Arch.
- CHOPSA . . . Complex HOMogenous Parallel System Arch.
- CIPP . . . Complete/Incomplete Parallel Processing
- OUPP . . . Object Update Parallel Processing
- FSP . . . File Simulation Processing

Fig.3.10 The Different Aspects of Parallel Processing

As mentioned before, one of the research objectives is to build a parallel KBZ OODBMS prototype using Meiko Computing surface. In the current prototype, as it will be discussed in the next chapters, the Simple Homogenous Parallel System Architecture has been chosen for implementation. Other aspects of parallelism are implemented too, including Parallel Background. Also, the File System Simulation process has been implemented to check the possibility of using parallel processing with filing system. The File System Simulation process is implemented in the form of Page Objects. The current prototype has been designed and tested in a single user environment. The

experimental results will be illustrated in chapter seven.

This chapter has also discussed the different types of Memory Objects allocation and deallocation. The current prototype assumes that all Memory Objects are loaded in the main memory using Demanded Object criteria. The prototype also groups all Memory Objects in a single Object Group.

CHAPTER 4

PKBZ Implementation : Object Class Representation

4.1. Introduction :

In the previous chapter, the different aspects of parallel processing with a PKBZ object oriented database management system are discussed. An example of each aspect is described in detail. Moreover, the advantages and disadvantages of the different aspects are illustrated too.

In the next three chapters, the design and implementation of the prototype PKBZ will be discussed. It has to be mentioned that the design and implementation phases are iterative processes. They are changed over time and according to the need. Accordingly, there are two versions of PKBZ. The initial prototype, PKBZ version-1, concentrates on the parallel implementation and is designed to investigate the functionality of the system. The system has been enhanced in PKBZ version-2 so that some experimentation can be carried out. In addition, some of the processes in version-1 have been modified in version-2 as a result of a review of the design. Thus, PKBZ version-2 supports the same functionality as version-1, but some processes are more sophisticated; in addition, some new functions have been added.

In this chapter and chapter 5, the discussion of the object class representation and functionality will be based on PKBZ version-1, while in chapter 6, the enhancements and modifications in version-2 will be discussed, together with the experimentation and results.

Moreover, the chapter will answer many questions which are relevant to any object oriented database system design and implementation. These questions are:

1. How are the different types of the PKBZ object class logically represented ?
2. How are the object classes physically stored ?
3. What is the system data ?
4. How is the system data logically and physically represented ?

5. How are the database files structured into the secondary storage ?

4.2. Logical Representation of the Object Class :

An object class, during the run time, is stored in a chunk of the main memory. This main storage is called Memory Object as defined in the previous chapter. Although, in the KBZ model, there are many object class types, in PKBZ version-1, all the different object classes, including Page Object, are mapped into the same Memory Object structure during the run time. The main reason is to simplify the design and the implementation phases. However, the system is modified in PKBZ version-2, in which each object class type is represented by a different Memory Object type.

A Memory Object is implemented in Occam as a single process. Inside this process, both the object class schema and the object instances are represented in the form of variables and arrays. The Memory Object data structure representation will be described later. The Memory Object communicates with the external system through message sending. The messages are sent to the Memory Object via channels. Fig.4.1. illustrates the Memory Object block diagram.

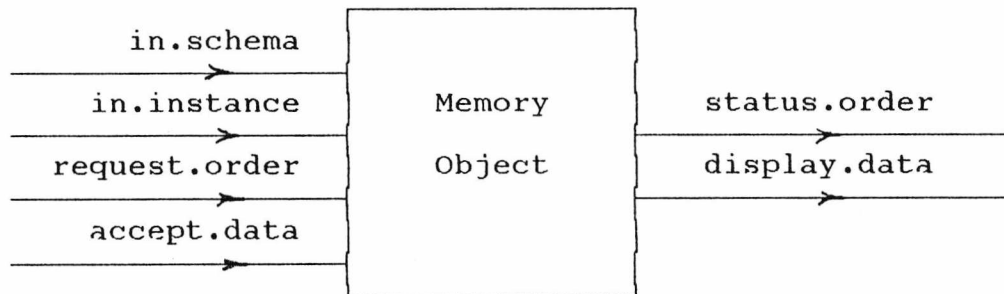


Fig.4.1. Memory Object Block Diagram

The Memory Object process can be represented as follows (the complete protocol definitions are illustrated in appendix D.) :

```
PROC Memory.Object (CHAN OF OBJECT.SCHEMA   in.schema   ,
                   CHAN OF INSTANCE         in.instance  ,
                   CHAN OF REQUEST.ORDER    request.order ,
                   CHAN OF ANY               accept.data  ,
                   CHAN OF INT              status.order ,
                   CHAN OF ANY              display.data )
```

```

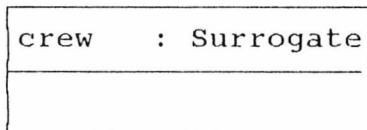
... Schema Variables Declaration
... Instance Variables Declaration
... Auxiliary Variables Declaration
... Auxiliary Process
SEQ
... Accept Schema From Channel in.schema
... Accept Instances From Channel in.instance
... Initialize and Prepare Auxiliary Variables
continue := TRUE
WHILE continue
... Accept Memory Object Message and Invoke The Corresponding Method
:

```

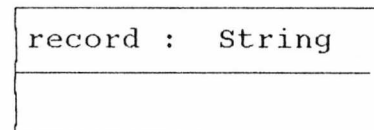
A Memory Object is created empty. It has no information. Then, a corresponding object class schema is sent through the "in.schema" channel, followed by the object instances through the "in.instance" channel. Hence, the Memory Object is ready to receive any further message through the "request.order" channel. The methods are stored and described inside each Memory Object. If any message is received, then the corresponding method is invoked inside each Memory Object. The status of the message is sent externally through the "status.order" channel. A Memory Object accepts any other information through the "accept.data" channel, while any data can be sent externally through the "display.data" channel.

The following KBZ object classes, which are described in chapter 1, together with the Page Object represented in KBZ notation will be used to illustrate the main data structures in the Memory Object implementation process :

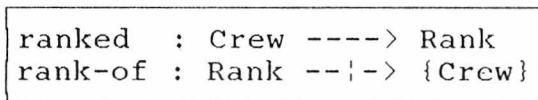
CREW



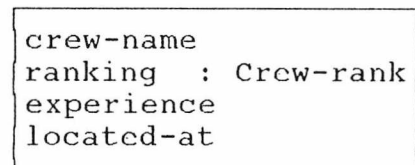
PAGE-OBJECT



CREW-RANK



STAFF



The Memory Object representation can be classified into the following :

1. Memory Object schema representation

2. Memory Object instances representation
3. Object methods and constraints representation

4.2.1. Memory Object Schema Representation :

The structure of the Memory Object schema does not depend upon the object class type. It contains the following data :

- a. Object Identification (Type, Surrogate, Name)
- b. Property names and types
- c. Inheritance information
- d. Constraints and functional properties
- e. Instances description (Instances type, Total number, ... etc)

The instance description is used to describe information related to the object instances. Such information will be described after the Memory Object instances representation has been discussed (subsection E in 4.2.2.)

The schema data structure representation is chosen to be minimal and simple to facilitate the design and the implementation procedures. The following schema variables are used to store the data values of the corresponding object schemata, while the complete description of the schema data structure is illustrated in appendix B :

A. Object Identification :

The following table illustrates an example of the complete object identification values of the CREW-RANK, STAFF, and PAGE-OBJECT object classes (see appendix B.1.):

Object Class	CREW-RANK	STAFF	PAGE-OBJECT
Component Name			
object.type.id	structural. type.int. string	entity. aggregatc. type	attribute. type.string
object.id	object identifier of CREW-RANK	object identifier of STAFF	Page Number in secondary storage
object.name	CREW-RANK	STAFF	PAGE-OBJECT

Appendix A.2. shows the complete description values of all object class type identifiers stored in "object.type.id". Each object class is identified uniquely by a surrogate value. This surrogate is generated by the system (see subsection 4.3.). The value of each surrogate is stored in the object identifier (i.e. "object.id").

B. Property Names and Types of The Object Class :

Although each object class type has a different data structure for the property names and types, a single data structure (see appendix B.2.) is used to accommodate all the object class types as shown below:

B.1. Basic Object Property Names and Types :

The following table illustrates the complete property names and types of the CREW and the PAGE-OBJECT object classes :

Object Class	CREW	PAGE-OBJECT
Component Name		
total.no.of.data.property	1	1
property.name[0]	crew	record
property.type[0]	Surrogate	String

The property type for basic object can be either :

- "Surrogate" (For entity set)
- "Integer" (For an attribute set of integer instance type)
- "String" (For an attribute set of string instance type)

B.2. Structural Object Property Names and Types :

The following table illustrates the complete property names and types of the CREW-RANK object class :

Object Class	CREW-RANK	Description
Component Name		
total.no.of.data.property	3	Number of stored data properties
property.name[0]	ranked	Primary relationship name
property.name[1]	rank-of	Inverse relationship name
property.type[0]	Crew	Domain of primary relationship
property.type[1]	Rank	Range of the primary relationship
property.type[2][0]	Y (Yes)	Primary relationship is (one-to-one)
property.type[2][1]	N (No)	Primary relationship is total function
property.type[2][2]	N (No)	Secondary relationship is (one-to-many)
property.type[2][3]	Y (Yes)	Secondary relationship is partial function

Although the property names and the property types of the structural object are completely different in structure, the table above shows how each relationship name with the corresponding domain and range can be stored completely in the system.

It has to be mentioned that the "property.type[2]" is used to store both the degree of relationship (one-to-one, one-to-many) and the existence characteristics (total, partial) of both the primary and the inverse relationships. Moreover, neither the domain nor the range of the inverse relationship are stored, since these are implicitly known from the primary relationship.

B.3. Entity Aggregate Object Property Names and Types :

The following table illustrates the complete property names and types of the STAFF object class :

Object Class	STAFF
Component Name	VALUE
total.no.of.data.property	4
property.name[0] property.name[1] property.name[2] property.name[3]	crew-name ranking experience located-at
property.type[0] property.type[1] property.type[2] property.type[3]	Crew-rank

As shown in the previous table, if the property type is declared, both the property name and the corresponding property type are stored in the same order (index) in the related array. But, if the property type is omitted, the property name will be stored only, since it implicitly indicates the property type.

As mentioned before, all the different object class types can be accommodated in the same data structure. However, there is a possibility to have a different data structure for each object type, but for simplicity, the system is implemented using a single data structure.

C. Inheritance Information :

Appendix B.3. describes the object class inheritance information data structure. The system is designed to allow a maximum fixed number of inherits and inherited by object classes (see appendix A.1.), since Occam language does not allow a variable array declaration. However, another data structure (e.g. linked list mechanism) can be implemented to accommodate a variable number of the inheritance information. But, for simplicity a maximum fixed number of array elements are used for the prototype implementation.

During the system running, the elements of the integer arrays "inherits" and "inherited.by" which are not used to reference an inheritance information are

initialized to zero values, while the corresponding elements of the string arrays "inherits.name" and "inherited.by.name" are initialized to space.

For example, the following table illustrates the complete stored inherits information of both CREW-RANK and STAFF object classes :

Object Class	CREW-RANK	STAFF
Component Name		
inherits[0]	CREW object identifier	CREW-NAME object identifier
inherits[1]	RANK object identifier	CREW-RANK object identifier
inherits[2]		EXPERIENCE object identifier
inherits[3]		LOCATED-AT object identifier
inherits.name[0] inherits.name[1] inherits.name[2] inherits.name[3]	CREW RANK	CREW-NAME CREW-RANK EXPERIENCE LOCATED-AT

On the other hand, the following table illustrates the complete inherited by information of the CREW-RANK object class :

Object Class	CREW-RANK
Component Name	
inherited.by[0]	STAFF object identifier
inherited.by. name[0]	STAFF

It has to be mentioned that the CREW-RANK object class does not store the inherited by object class information until STAFF object class is created. That is, when the STAFF object class is created, the system adds the STAFF object class to the CREW-RANK inherited by information.

STAFF has no inherited by information and therefore contains only the initialized

values.

D. Constraints and Functional properties :

The data structure of the constraints and functional properties have been designed (see appendix B.4, and B.5), but the real implementation has not been considered for the reasons described in subsection 4.2.3.

4.2.2. Memory Object Instances Representation :

It is difficult to have a single representation for all different object class instances, although all the various object class types have the same logical schema representation. So, each object type has its own object instance structure representation.

It has to be mentioned that the PKBZ version-1 prototype allows a fixed number of object instances to be stored only; the current maximum value is the "max.number.of.objects" (see appendix A.1). This variable is set to 50 per Memory Object in PKBZ version-1. In the PKBZ version-1, the complete instances are stored in the Memory Object during the system life. However, the PKBZ version-2 allows more instances to be stored in which the instances are swapped between the Memory Object and the corresponding secondary storage.

Instances are stored in Memory Object in the form of integer and string arrays. For example, an entity set's instances are stored as an integer array. A structural object's instances are stored in two different integer arrays if the range of the primary relationship is an integer, or in an integer array and string array if the range of the primary relationship is a string. The data structure of attribute set instances is designed (see appendix C.1., and C.2.), but attribute set instances are not stored independently; they are only stored in the corresponding structural objects in the array representing the primary relationship range. But, there are two exception cases :

- i. Page Object instances : The instances of Page Object have to be stored.
- ii. System Objects instances : The System Objects will be described in subsection

4.3.

The only derived object type currently implemented is the entity aggregate which is stored as an integer array which contains the surrogate values of the entities on which the entity aggregates are based. This is explained in more detail below, together with the examples:

A. Basic Memory Object Instances Representation :

The CREW entity set shown below in its KBZ notation is used to store the unique surrogate of each crew member in the AirLine Transportation Example.

CREW

crew	: Surrogate

The data structure of the basic Memory Object instances representation is illustrated in appendix C.1. For example, the instances of crews are stored in the CREW entity set as follows:

Object Class	CREW
Component Name	
id.instance[0]	12
id.instance[1]	13
id.instance[2]	14
...	..
...	..
...	..
...	..

The instances values of the CREW or any other entity set are generated sequentially by the system using the surrogate value in "instance.description[4]" (see subsection E.).

B. Structural Memory Object Instance Representation :

As described before, the domain of the primary relationship in the structural

object is always an entity set, then the domain value can be represented by an array of integers. Similarly, if the range is an integer too, another array of integer is used to represent the range. But, if the range is a string, it is represented by an array of string.

For example, let us consider the following KBZ object classes which are described in chapter 1 :

BUS-TYPE

```
has-type: Plane----> Type
type-of : Type ---->{Plane}
```

BUS-CAPACITY

```
: Plane----> Capacity
: Capacity--->{Plane}
```

BUS-AGE

```
: Plane---> Hour
: Hour ---> {Plane}
```

The data structures of the Memory Object instance representations are illustrated in appendix C.3. The following tables illustrate an example of the data storage values in the different KBZ structural objects :

Variable Name	id.instance[i] (Corresponding to domain PLANE)	name.instance[i] (Corresponding to range TYPE)
i		
0	11	Concorde
1	12	Boeing 747
2	15	Boeing 747
3	17	Tristar
4	25	Concorde
.
.
.

BUS-TYPE instances

Variable Name	id.instance[i] (Corresponding to domain PLANE)	n[i] (Corresponding to the range CAPACITY)
i		
0	11	400
1	12	350
2	15	300
3	17	450
4	25	500
.
.
.

BUS-CAPACITY instances

Variable Name	id.instance[i] (Corresponding to domain PLANE)	n[i] (Corresponding to range HOUR)
i		
0	11	2080
1	12	1158
2	15	1500
3	17	2099
4	25	1112
.
.
.

BUS-AGE instances

The data values are stored corresponding to the definition of the primary relationship in the structural object. The same index [i] on both arrays (domain and range) is used to link both the domain and the range together to form a single instance.

C. Derived Memory Object Instances Representation :

As mentioned before, the only derived object type currently implemented is the entity aggregate. Since, the domain of all object classes which are defined inside the schema declaration (i.e. the common domain of the inherit structural objects) of any entity aggregate is the same entity set, then the common surrogate values are the only instances which are stored inside the entity aggregate object class.

For example, let us consider the AIRBUS entity aggregate object in the AirLine Transportation Example :

AIRBUS

bus-type bus-capacity bus-age

The common domain of the above three structural objects is the PLANE entity set. Then, the surrogate values of the common domain PLANE are the only instances which are stored in the AIRBUS entity aggregate object as follows (see appendix C.4.) :

Object Class	AIRBUS
Component Name	
id.instance[0]	11
id.instance[1]	12
id.instance[2]	15
id.instance[3]	17
id.instance[4]	25
...	..
...	..

AIRBUS stored instances

The main advantages of storing the surrogate values only of the common domain are :

1. The minimal storage of the entity aggregate is achieved in both the main memory (i.e. Memory Object) and in the secondary storage.
2. Thus, the whole entity aggregate instances can be instantiated from the related inherits objects, during the run time, in parallel using "Parallel Data Distribution" or "Parallel on a Single Transputer" as described in chapter 3. It has to be mentioned that PKBZ instantiates the entity aggregate instances using "Parallel On Single Transputer". The instantiated instances will be as shown in the following table :

Structural Obj.	Domain Value (PLANE)	BUS-TYPE (Primary range)	BUS-CAP. (Primary range)	BUS-AGE (Primary range)
Index [i]				
0	11	Concorde	400	2080
1	12	Boeing 747	350	1158
2	15	Boeing 747	300	1500
3	17	Tristar	450	2099
4	25	Concorde	500	1112
.
.
.

AIRBUS Instantiated Instances

D. Page Object Instance Representation :

Appendix C.5. describes the data structure of the Page Object instance. Each Page Object is used to store the corresponding Memory Object Instances (MOI) in the main memory. So, the Memory Object Instances can be distributed in the Page Object as shown in Fig.4.2 :

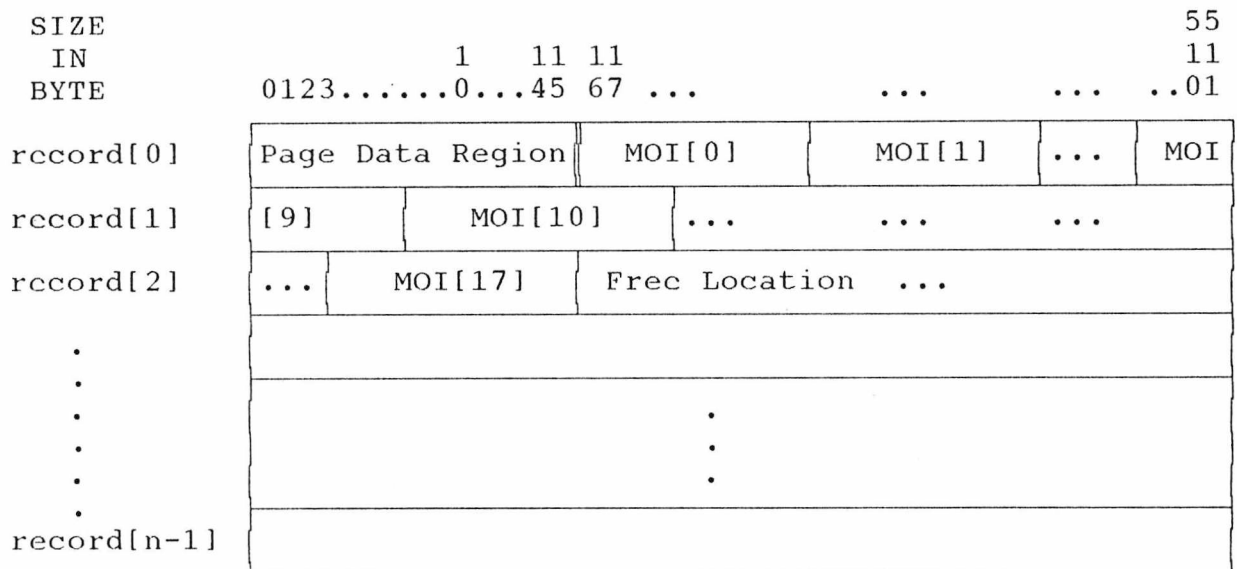


Fig.4.2. Memory Object Instances Distribution In The Page Object

The Page Object Structure is chosen to reflect the secondary storage file structure which is in terms of 'records'. Thus, each Page Object instance is a record, and a Page Object consists of "n" records. Each record is a string. The size of each record is set to "fileSys.maxDataByte" (FSMDB see appendix A.1) which is equal to the maximum

record length size available in the Occam language, and it is also equal to the record width of the secondary storage file which contains the PKBZ database. The FSMDB is chosen by Occam library to be 512 bytes. In the PKBZ system, the number of Page Object instances per Page Object ("NO.REC.IN.PAGE" see appendix A.1.) is set to 5 (i.e. $n = 5$). This value is chosen arbitrarily.

The first 16-bytes in record[0] are used to store information about the Page Object instances. During the run time, these 16-bytes are transformed into a single 8-element integer array, called "Page.Data.Region". The function of each element in this array is described in appendix C.6.

The rest of the record[0], and the other records in the Page Object instances are used to store the corresponding instances of the related object class, as shown in Fig.4.2.

The following table illustrates an example of the data values corresponding to the "Page.Data.Region" in Fig.4.2. :

Component Name	VALUE	DESCRIPTION
Page.Data.Region[0]	2	Number of stored records
Page.Data.Region[1]	18	Number of stored MOI
Page.Data.Region[2]	16	Current Page Number
Page.Data.Region[3]	0	Previous Page Number
Page.Data.Region[4] to Page.Data.Region[7]	zeros	Reserved

It is assumed that this Page Object is corresponding to the physical page 16 in the secondary storage ("Page.Data.Region[2]").

The Memory Object Instances are stored consecutively. Even, if the record does not accommodate the complete instance, the rest of the instance string will be extended to the next record (e.g. MOI[9], in Fig.4.2). By this mechanism, all records are used and no wasted space exists at the boundary.

As described before, in PKBZ version-1, only one Page Object is allocated to each

Memory Object. The Page Object is sufficient enough to accommodate the "max.number.of.objects" in the Memory Object. However, in PKBZ version-2, more than one Page Object is used to accommodate the Memory Object Instances. If an instance does not fit at the end of the page, then the whole instance value will be accommodated to the next page. That is, it will not be partitioned between the two consecutive pages. Then, the only gap that exists in the Page Object instance is at the end of last record.

E. Instance Description :

As mentioned before, the instance description forms part of the Memory Object schema. It is used to describe information related to the object instances. The following table illustrates the data structure and the corresponding values of the instances description of CREW, BUS-TYPE, BUS-CAPACITY, and AIRBUS in the Memory Object (see appendix B.6.):

Object Type	Entity Set	Structural Object		Entity Aggregate
Object Class	CREW	BUS-TYPE	BUS-CAPACITY	AIRBUS
instance.description				
0	0	0	0	0
1	p1	p2	p3	p4
2	pdr+b1*i1	pdr+b2*i2	pdr+b3*i3	pdr+b4*i4
3	i1	i2	i3	i4
4	s	0	0	0
5	0	0	0	0
6	0	0	0	0
7	int	int.string12	int.int	int

The "instance.description[1]" indicates that the instances of the CREW, BUS-TYPE, BUS-CAPACITY, and AIRBUS are stored in the database file pages p1, p2, p3, and p4 respectively (see subsection 4.5.).

The zero value in "instance.description[0]" describes that there is no previous instance page in the secondary storage database file. In general, in PKBZ version-1 only one instance page is allocated to each object class to simplify the implementation process.

Moreover, the "instance.description[3]" keeps track of the number of Memory Object Instances. It is assumed that the number of object instances currently stored in the CREW, BUS-TYPE, BUS-CAPACITY, and AIRBUS are i1, i2, i3, and i4 respectively. This number is updated automatically, during the system execution, according to the message applied to the corresponding Memory Object.

The "instance.description[4]" stores the current surrogate value. This value is updated by the system only. It is set to "s" in the CREW entity set. But, it is clear that it has no meaning in the BUS-TYPE, BUS-CAPACITY and AIRBUS, since no surrogate is generated, then the corresponding value is set to zero.

The "instance.description[7]" represents the instance type. For example, the instance type integer "int" is stored in both CREW and AIRBUS. While, in the BUS-TYPE, "int.string12" describes the domain as an integer and the range as a string of length (size) 12 bytes, in the BUS-CAPACITY, "int.int" illustrates that both the domain and the range have type integer. The system assumes that "int" is corresponding to int16 in Occam language which can be represented by two bytes. "int32" is not considered in the prototype implementation.

The "instance.description[2]" points to the free location within the corresponding Page Object (see Fig.4.2.). This location is calculated by the formula "pdr + b * i", where "pdr" is the size in bytes of the "Page.Data.Region", "b" is the size in bytes of the corresponding logical instance (see appendix A.4.), and "i" is the number of the current stored Memory Object Instances in the corresponding Page Object.

Both the "instance.description[5]" and "instance.description[6]" are not currently used. They are reserved for future system modifications.

It has to be mentioned that the object instances are stored in sequence. For simplicity no index mechanism is performed in the current prototype. But, in future



implementations, an index mechanism could be carried out for fast retrieval. Such index mechanism could be stored in the instance description.

4.2.3. Object Methods and Constraints Representation :

As described in Memory Object process, when the Memory Object receives both the object schema and the related object instances, then Memory Object is ready to accept any further message. When a message is received through the "request.order" channel (Fig.4.1.), the corresponding method is invoked. The structure of such object methods will be described as follows :

```
request.order ? CASE
  message.1 ; ... ; ...
  SEQ
  method.1
  message.2 ; ... ; ...
  SEQ
  method.2
  ...
```

The CASE statement is used to select one method from the available different methods. The "message.1", and "message.2" are the tag protocol of the message.

The complete PKBZ protocol definitions are given in appendix D. Further, the Memory Object Messages data structures are illustrated in appendix E.

It has to be mentioned that the current PKBZ system implementation enables only system methods. The user's methods are not implemented, since methods are corresponding to instructions or procedures, as shown above, and there is no way to append user's methods to the system methods after these procedures are compiled. So, a compiler or an interpreter is needed to transform user's instructions into an executable form. So, the data structure of the functional properties (appendix B.5.) is designed to allow further modification in future. The end user can define his or her functional properties. Then, the system can transform these functional properties into an executable form and store this information inside a class object.

There is a possibility that the end user could design his or her functional properties using the Occam code and compile it. Then, a set of compiler library

procedures called "Dynamic Code Loading" [INMOS 89] could read this section of compiled code and execute it. Such modifications could be implemented in future research.

Similarly, constraints are not yet implemented, for the same reason. But, the data structure of the constraints is designed (appendix B.4) for future modifications.

4.3. System Object Classes :

The system has to know the complete information about each user's object class. Such information is called System Data. Since the user's data is stored in the form of object classes, then it is best to store the system data in the form of object classes too. Fig.4.3., and Fig.4.4. show the structure of the system object classes representation in graph notation and KBZ notation respectively.

The functions of each system object class will be illustrated as follows:

A. OBJECT-ID Object Class :

The "OBJECT-ID" (Object Identification) is an entity set. It is responsible for generating a unique surrogate for each object class in the system, since each object class is identified by a unique surrogate. Surrogates are generated in sequence. The object instances of the "OBJECT-ID" entity set are all the object classes surrogates. That is, when the "OBJECT-ID" entity set is initialized, it has only five surrogates from zero to four corresponding to the five system object classes.

B. OBJECT-NAME Object Class :

The "OBJECT-NAME" is an attribute set. It stores all object classes names in the system. The object class names are stored as object instances in this attribute set. That is, when the "OBJECT-NAME" attribute set is initialized, it has only five names of the system object classes : OBJECT-ID, OBJECT-NAME, OSHEMA, HAS-ONAME, and HAS-OSHEMA.

It has to be mentioned that although the attribute set instances are not stored for the user's data, the system allows such storage for system data for purpose of designing and debugging the system during the system implementation.

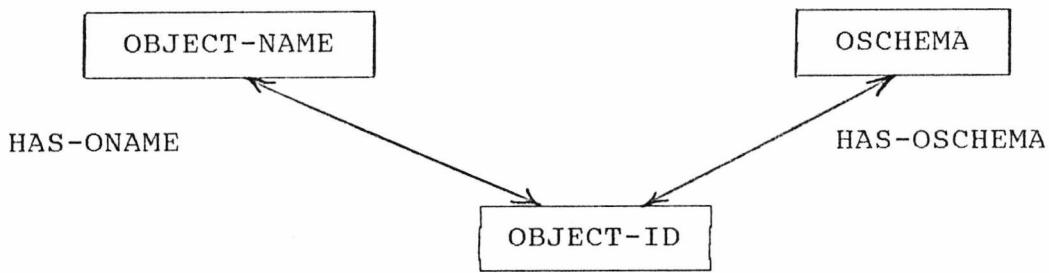


Fig.4.3. System Object Classes Description

OBJECT-ID

object-id : Surrogate

OBJECT-NAME

object-name : String
length(object-name) <= 12

OSHEMA

oschema : Integer

HAS-ONAME

has-oname : Object-id ---> Object-name
named : Object-name ---> Object-id

HAS-OSHEMA

location : Object-id ---> Oschema
located-at: Oschema ---> Object-id

Fig.4.4. System Object Classes Description in KBZ notation

C. OSHEMA Object Class :

The schema of all object classes are stored separately in a special location in the secondary storage database file (see subsection 4.5). This position of each object class schema is stored in the "OSHEMA" (Object SCHEMA) attribute set. When the "OSHEMA" is initialized, it has five integer numbers. Each one is corresponding to a unique position of each system object class schema.

D. HAS-ONAME Object Class :

The "HAS-ONAME" (HAS Object NAME) is a structural object. When a new object class is created, the "OBJECT-ID" entity set generates a new surrogate. This new surrogate along with object class name is stored as a single object instance in the "HAS-ONAME" structural object. So, when the "HAS-ONAME" structural object is initialized, it has five object instances. Each object instance is corresponding to one of the system object classes.

E. HAS-OSHEMA Object Class :

The "HAS-OSHEMA" (HAS Object SCHEMA) is a structural object. Its object instances are the object class surrogates along with the position of each object class schema in the secondary storage database file. The "HAS-OSHEMA" structural object neither allows two object class schemata to be stored in the same location nor permits two object schemata to be overlapped. So, when the "HAS-OSHEMA" structural object is initialized, it contains five object instances. If a new object class is created, the surrogate will be stored along with the new object schema position.

** Advantages :*

1. The main advantage of representing system data in the form of KBZ object classes is that the design and implementation procedures are expected to be easier, since there is no need to design and implement two different structures one for the system data and another one for the user's object classes.

2. Both system data and user's data are represented in the form of the object classes so that any further modifications in the object class structure may be reflected in both of them.

4.4. Physical Representation of the Object Class :

A Memory Object is represented by schema and object instances. So, the simple way to store such representation into the secondary storage database file is to aggregate all the corresponding variables into a single record. Then, this record is written into the secondary storage database file. Unfortunately, such physical storage in the form of record is not applicable in Occam, since Occam does not support the record definition [INMOS 88.b.].

So, one possible way of overcoming such difficulties is to design a routine which transfers the Memory Object structure into a single string. That is, Memory Object is transformed into a logical record in the form of a single string. Then, this single string is written into the database file.

One of the simplest methods of transformation is to transfer each integer variable into an equivalent character string. Then, all strings are collected together to form a single string. The Fig.4.5. shows the block diagram of such transformation.

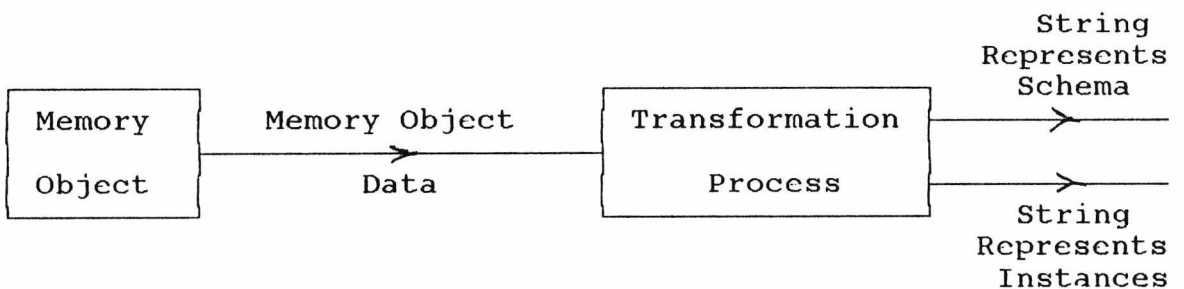


Fig.4.5. Memory Object Transformation Process

The transformation mechanism is carried out in two phases. In the first phase, the object instances are transformed. Then, in the second phase, the object schema is transformed. The following subsections illustrate the transformation mechanism in detail :

4.4.1. Instances Memory Object transformation :

The following code illustrates the instances conversion :

```
PROC Convert.update.objects.instances(... , ... ,      ...)  
... Variable declaration  
SEQ  
... Accept instance type (i.e. instance.type) of the object class  
— (i.e. instance.description[7] variable (see subsection E. in 4.2.2))  
CASE instance.type  
  int      — Integer instance  
    PAR  
      ... Invoke "intermediate.create.int.instance" Process  
      ... Accept number of records + Page  
  string12 — String12 instance  
    PAR  
      ... Invoke "intermediate.create.string.instance" Process  
      ... Accept number of records + Page  
  int.string12 — int + String12 instance  
    PAR  
      ... Invoke "intermediate.create.structural.instance" Process  
      ... Accept number of records + Page  
  int.int   — int + int instance  
    PAR  
      ... Invoke "intermediate.create.int.structural.instance" Process  
      ... Accept number of records + Page  
:
```

There are four different transformation processes corresponding to the four different instance types allowed in PKBZ system (see appendix C.). The description of each process will be as follows :

A. *intermediate.create.int.instance* Process :

The following code illustrates the function of the process :

```
PROC intermediate.create.int.instancecc(... , ... ,      , ... )  
... Variables declaration  
SEQ  
  in ?   size:=instance FROM 0 FOR size]  
  st := 16  
  SEQ i = 0 FOR size  
    SEQ  
      convert.int.string2(instance[i] , [rec FROM st FOR 2])  
      st := st + 2  
  convert.array.to.page(rec , page)  
  out ! INT( (st - 1) / FSMDB ) ; page  
:
```

The first 16 bytes are left to store the "Page.Data.Region". Then, the integer instance array is transformed into a single string array "rec". The "rec" array will be

as shown in Fig.4.6. :

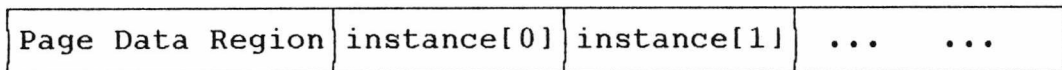


Fig.4.6. Transformation of Integer instances into a single String

Then, the single array is transformed into a page format (i.e. two dimensional array). The number of stored records per page is sent together with the page. This page structure has the same structure as the Page Object instance.

B. intermediate.create.string.instance Process :

Although string instances in the attribute set are not stored, this process is designed at the beginning of the system implementation for the system objects to facilitate the design and the debugging phases.

The following code illustrates the process :

```
PROC intermediate.create.string.instance(... , ... ,      , ... )
  ... Variables declaration
  SEQ
  ... Accept string instances and the size
  ... Store each instance in the corresponding position
  ... Convert the single array into a page
  ... Send the number of records used with the page
:
```

C. intermediate.create.structural.instance Process :

The following code illustrates the process :

```
PROC intermediate.create.structural.instance(... , ... ,      , ...)
  ... Variables declaration
  SEQ
  ... Accept integer + string instances and the size
  ... Convert each integer domain instance into two bytes string
  ... Store each instance in the corresponding position
  ... Convert the single array into a page
  ... Send the number of records used with the page
:
```

Each integer domain in the structural instance is converted into a string. Then, the string domain is concatenated with the string range to form a single instance as shown in Fig. 4.7. :

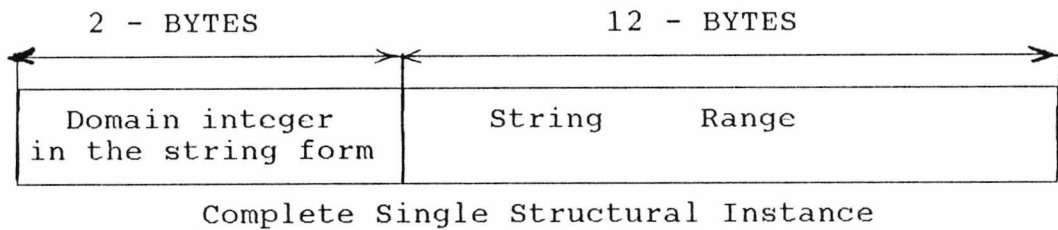


Fig.4.7. Transformation of Structural instances in the form of integer and string into a single String

D. *intermediate.create.structural.int.instance* Process :

The following code illustrates the process :

```

PROC intermediate.create.structural.int.instance(... , ...)
... Variables declaration
SEQ
... Accept integer + integer instances and the size
... Convert each integer into two bytes string
... Store each instance in the corresponding position
... Convert the single array into a page
... Send the number of records used with the page
:

```

Each structural instance is converted into a single string. This single string will be as shown in Fig.4.8. :

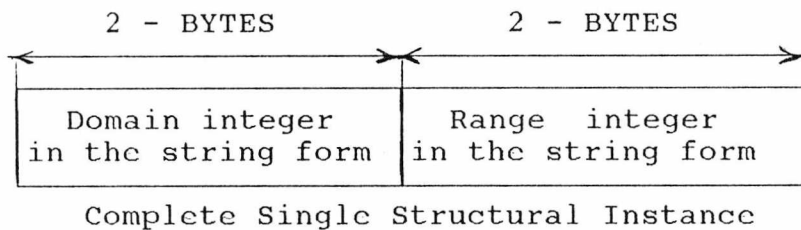


Fig.4.8. Transformation of Structural instances in the form of domain integer and range integer into a single String

After the instances are transformed into a page, each page of instances will be sent to the corresponding Page Object in the SMS module. This mechanism is carried out using Object Update Parallel Processing. Then, each Page Object is transformed into the secondary storage using the File Simulation Processing.

4.4.2. Schema Memory Object Transformation :

When transforming the Memory Object schema into its Page Object representation, some schema data are suppressed. These suppressed data are :

1. Object name
2. Inherits object names
3. Inherited by object names

These suppressed data have reference identifiers. So, during the loading process of any object class from the secondary storage to the main memory, the corresponding identifiers of the suppressed data are used as an index to get the suppressed data. By this mechanism, the storage space is optimal. The following code illustrates how the object schema is transformed into a single string :

```
PROC intermediate.create.object( ... , ... , ... , ... )
  ... Variables declaration
  SEQ
  PAR
    ... Convert object.type.id into string
    ... Convert object.id into string
    ... Convert property names and property types into a single string
    ... Convert inherit and inherited by identifiers into a single string
    ... Convert constraints and functional properties into a single string
    ... Convert instance description into a single string
  ... Aggregate all the previous strings into a single string
  ... Get the total size of the aggregated single string
  ... Convert the total size into a string too
  — and prefix it at the beginning of the aggregated string
  ... Send the string represents the object schema together with its length
:
```

As described above, each part of the schema is changed into a string, then each string is aggregated together with its length to form a single string.

Thus, the Memory Object is transformed into two different strings which are corresponding to the object schema and the object instances respectively. Such transformation is suitable for the database file structure which will be discussed in the next subsection. To transform the data from physical representation into a logical representation, the reverse transformation is carried out.

However, if Occam supported the record format representation, neither the

transformation mechanism, nor the inverse operation would need to be carried out.

4.5. Database File Structure :

To simplify the design and the implementation phases, the database file is chosen to be a single file. Its access mechanism is a random access mode [INMOS 88.b.]. It is constructed of fixed length records. The fixed record length is set to `fileSys.max.DataBytes` (FSMDB), as defined in Occam library (see appendix A.1.). A number of records are grouped together to form a single page. This number is chosen to be an arbitrary number, and it is set to "5", as defined before in Page Object. The page is the unit of access (reading or writing) for the database file.

The database file is divided mainly into three groups. Each group stores special information related to this group. Each group is allocated a number of pages. This number is variable according to the function of the group. Fig.4.9. represents the internal structure of the database file.

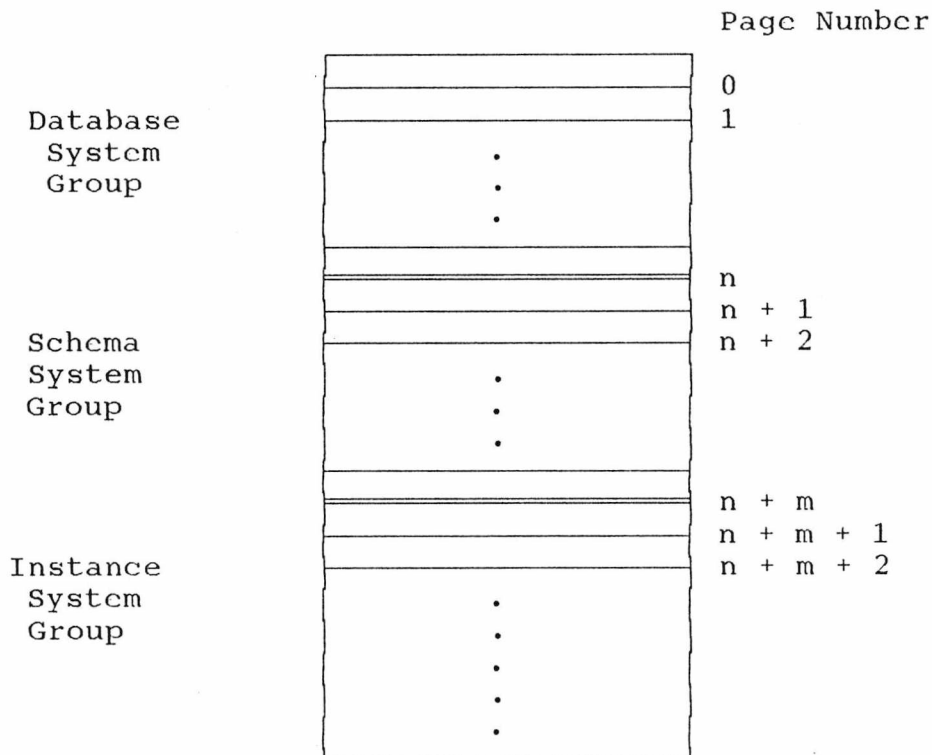


Fig.4.9. Database File Structure

The three groups are :

- a. Database System Group
- b. Schema System Group
- c. Instances System Group

The following subsections describe each group in detail. But, the first group will be described at the end, since its data depends upon the second and the third group.

4.5.1. Schema System Group :

The schema system group is the second part of the database file. It consists of "m" pages. The function of this group is to accommodate the schema of all object classes in the database. When a new object class is created or when any Memory Object is transformed into physical representation, the string corresponding to the object schema is collected from all object classes, then they are transformed into the form of records with a fixed length size. Fig.4.10. describes the internal structure of the Schema System Group :

SIZE							55
IN	1	2	3	33			11
BYTE	0123...	0..0..0	12...01

record[0]	schema[0]	schema[1]	schema[2]	...	sche
record[1]	ma[i]	schema[i+1]
record[2]	...	schema[n]	Free Location	...	
.					
.					
.					
.					
.					
record[n]					

Fig.4.10. Schema System Group

When the schema string does not fit into a single record, it is expanded into the

next record (e.g. schema[i]), and so on, until all strings are stored. Then, a fixed number of records forms a single page. Hence, the whole page is stored into the schema system group. The number of pages in the schema system group in both PKBZ versions is set to one. However, "m" can be any other value in future.

The position of each schema object class within this page is stored in the "HAS-OSHEMA" structural object, as described before. Then, the system can know the exact position of any object schema in the system. Once the position is known, the schema is retrieved first, then the object instances page location can be known from the instances description data.

4.5.2. Instances System Group :

The instances system group is the last group in the database file. It stores all the object instances in the form of pages. Each page is used to store the instances of a single object class. As mentioned before, the data structure of the page in the Instance System Group has the same data structure as the instances in a Page Object. That is because, instances in a Page Object fits into a single page in the database file.

If more than one page is needed for storage of any object class, the next page number is stored in the previous page corresponding to this object class and so on. So, a chain of pointers is used to link all the related pages together. That is, the instance pages corresponding to a single object class need not be consecutive. Moreover, in PKBZ version-2, the system allows the number of pages allocated to any object class to be variable, since no one can know in advance the exact number of instances for each object class. But, in PKBZ version-1, each object class allocates a single page in the secondary storage database file.

There is a possibility that each object schema and the corresponding instances could be stored together into a single page. But, when the number of object instances increases and more than one page is needed to accommodate the object instances, the structure of the other pages would be different from the first one, since only the first page would accommodate the schema structure and there is only one schema structure

per object class. That is why the system separates object schema and its instances in order to have homogenous object instances page structure. This homogeneity is reflected in the design of the Page Memory in File Simulation Processing.

On the other hand, to minimize the mechanical head movement on the system disk when a page instance is required to be read, the system reads the first record in the page first (see Fig.4.2) ¹, then the system knows the number of records in this page which is stored in the first record (i.e. Page.Data.Region[0]). Although, the page is the unit of reading or writing, the number of records stored prevents the unnecessary reading of empty records, if any. That is, the unnecessary mechanical motion of the head on the secondary storage disk. Moreover, the same mechanism is used during the writing process.

4.5.3. Database System Group :

The database system group is used to store information about the database file itself. Such information is important during the system life. The database system group stores the total number of used pages, the position of used and unused pages in the database file, the pages allocated to each group, ... etc.

Appendix F. shows the logical data structure of the information (i.e. sys.info array) which are transformed into physical data in the form of a page and stored in the Database System Group.

The following table illustrates an example of the "sys.info" array values :

Component Name	VALUE
sys.info[0]	7
sys.info[1]	0
sys.info[2]	27
sys.info[3]	2
sys.info[4]	31

continue to next page

¹ The database file page has the same structure as Page Object Instances

Component Name	VALUE
sys.info[5]	7
sys.info[6] to sys.info[12]	page numbers used
sys.info[13] to sys.info[50]	zeros

The "sys.info[5]" stores the number of used pages in the Instances System Group. For example, after the creation of the Database System Group page, the Schema System Group page, and the five system object classes in the Instances System Group, this value is 7. Then, the following 7 elements in the array (i.e. sys.info[6] to sys.info[12]), store the actual page numbers used in each case.

The next available page in the Instance System Group is stored in sys.info[0]. This value is used by the system when a new page is required to be allocated. When this page is used, the related values in the "sys.info" array are updated automatically by the system to reflect the transaction.

The sys.info[3] indicates that only three records in the Schema System Group are used (i.e. record[0], record[1], and record[2]) (see Fig.4.10.), and sys.info[4] indicates the position of the free location in the third record. This information is used to accommodate any new schema when a new object class is created.

The sys.info[1] indicates that one record (i.e. record[0]) is used to store "sys.info" array itself in the Database System Group, while sys.info[2] indicates that the available free location in this record is location 27. That is, the current 13 elements are stored in two bytes each. This information is necessary to know the "sys.info" array size itself in the secondary storage. The rest of the "sys.info" (i.e. sys.info[13] to sys.info[50]), which are not used, are initialized to zero during the system execution.

In general, this database file structure is not unique. There are many other possible structures. For example, the system can store each of the three previous groups

into three separate files.

4.6. Conclusion :

By designing the Memory Object as described in subsection 4.2., the concept of Object which is described in chapter 1 is satisfied quite well, since each Memory Object is viewed as a complete entity in itself. Further, the internal representation of data is completely hidden. Moreover, object class schema and object instances data are logically integrated together inside each Memory Object. Such integration is not supported by the currently available non object oriented databases.

In addition, data and related methods used to access or modify any Memory Object are encapsulated together within the Memory Object. Such encapsulation provides a way that the system can store the semantic aspects along with data.

By using message sending, the internal data structures and data inside the Memory Object are totally isolated. Hence, Memory Object modification and enhancement internally is relatively safe.

Although the structural object class has different property names and types structure than basic object and entity aggregate object, a single data structure is designed and implemented to be used for all different object classes.

The entity aggregate object instance structure corresponds to the tuple structure in the relational database system. The tuple structure may change from one entity aggregate to another, so it is not easy to get a standard structure. But, a standard data structure is designed for all entity aggregate instances irrespective to their tuple structure. Such a standard structure cannot be achieved in the relational database system. But in PKBZ, only the surrogate values are stored, and the corresponding instances data can be instantiated from the related structural object instances in parallel.

The system data is designed and implemented in the form of objects as with user's data to reflect any enhancement that may occur in the system and to minimize the design and implementation.

The lack of record definition in the Occam language is overcome by designing a conversion mechanism between the logical data representation and the physical data representation, and vice versa.

The storage of the Memory Object data in the secondary storage is optimal, since the data that have identifiers are suppressed before it is stored into the corresponding secondary storage and it can be retrieved when it is needed.

The database file is designed to be homogenous in the data structure. This homogenous nature is reflected in the Page Object. The database file is also designed to be optimal, since no gap exists at each record in the database file, except at the boundary of last record in each page.

The system stores the detailed structure of each page in the secondary storage to prevent the unnecessary mechanical motion of the head on the secondary storage.

Although Database System Group stores complex information related to different groups in the database file, the data structure is designed as a simple integer array.

CHAPTER 5

PKBZ Implementation : Messages

5.1. Introduction :

In this chapter, the distribution of the basic three modules of the PKBZ across different transputers are illustrated. Moreover, the hardware channel functions will be described.

Since the operation in the object oriented database management system is performed via message sending, each available message in the system will be discussed and how this message is executed in parallel in each module across the different transputers.

5.2. The PKBZ Version-1 General Block Diagram :

In general, as described in chapter 3, there are three different parallel system architectures. But, only one type of these architectures has to be chosen. During the designing and the implementation phases, no one can know in advance the detailed structure of each module, the internal processes (subroutines) which are required. So, grouping processes together and mapping each group into a different transputer cannot be achieved at the beginning of the design phase. Moreover, it is better to start with a simple design. Then, the design can be adapted to be more sophisticated over time according to the achievements. Thus, both the Complex Homogenous Parallel System Architecture and the Hybrid Parallel System Architecture are not taken into consideration, in this prototype, for the above reasons.

In this chapter, the discussion will be focused on the Simple Homogenous Parallel System Architecture. That is, each module of the PKBZ OODBMS will be mapped into a different transputer. Consequently, the system consists of only three transputers according to the three basic modules (EMS module, OMS module, and SMS module respectively). Fig.5.1. illustrates the actual distribution of the PKBZ version-1 modules,

and the wiring diagram among the transputers.

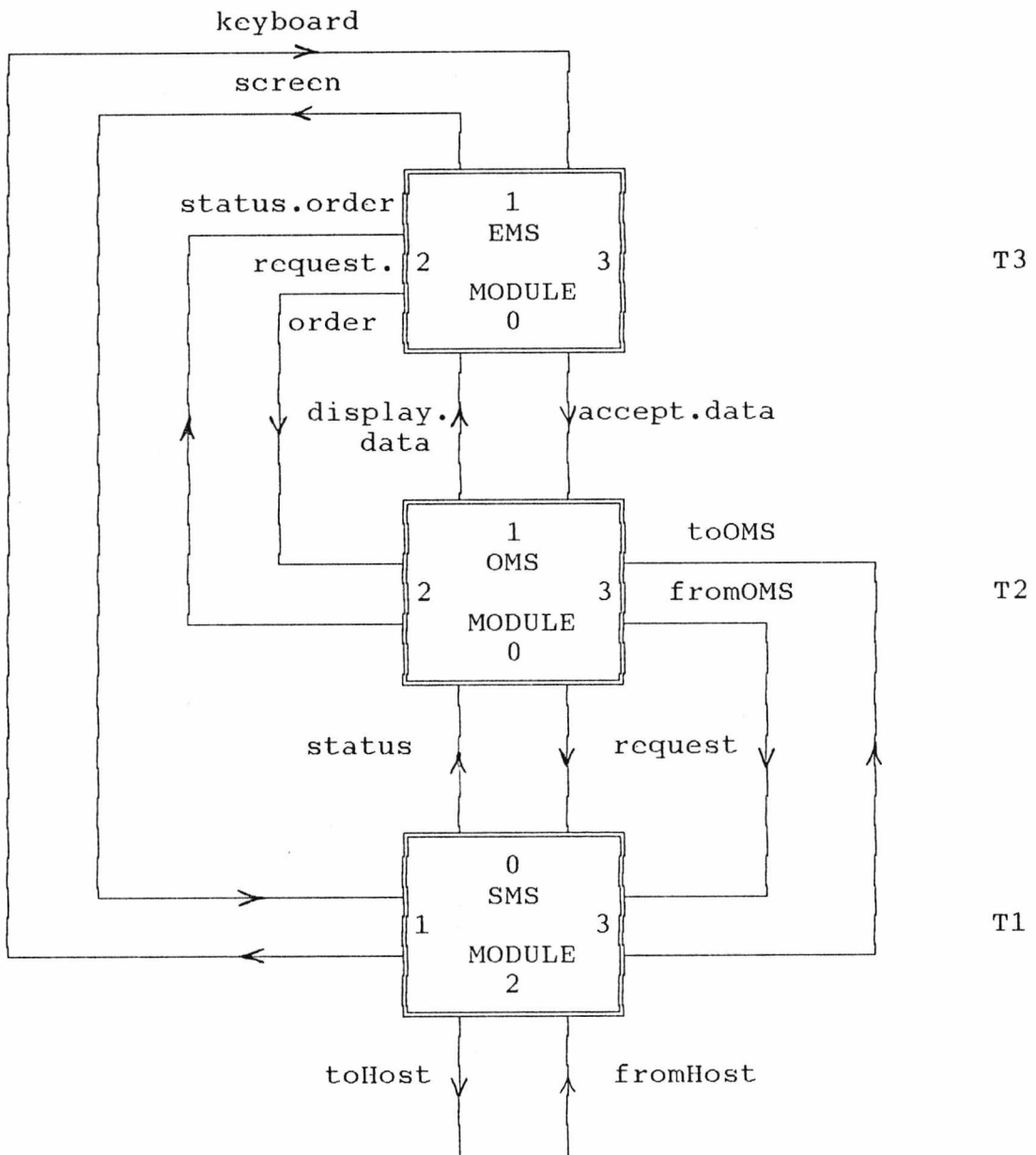


Fig.5.1. PKBZ Version-1 OODBMS Modules Distribution

It has to be mentioned that each hardware channel has its own type protocol. This type protocol depends upon the functions of each hardware channel in the system. The distribution of the simple Homogenous Parallel System Architecture can be represented as follows ¹:

¹ The indentation is more than two spaces to clarify the Occam code

```

-- Channel declaration
CHAN OF ANY          keyboard, screen, toHost, fromHost:
CHAN OF REQUFST.ORDER request.order , request      :
CHAN OF INT          status.order  , status        :
CHAN OF ANY          display.data  , accept.data   :
CHAN OF REQUEST.ORDER toOMS        , fromOMS       :
-- Transputers Allocation
PLACED PAR
  PROCESSOR 1 T8
    PLACE request      AT in.link.0:
    PLACE status       AT out.link.0:
    PLACE screen       AT in.link.1:
    PLACE keyboard     AT out.link.1:
    PLACE fromHost     AT in.link.2:
    PLACE toHost       AT out.link.2:
    PLACE fromOMS      AT in.link.3:
    PLACE toOMS        AT out.link.3:
    SMS(fromHost, toHost, keyboard, screen, request, status, fromOMS , toOMS )
  PROCESSOR 2 T8
    PLACE status       AT in.link.0:
    PLACE request      AT out.link.0:
    PLACE accept.data  AT in.link.1:
    PLACE display.data AT out.link.1:
    PLACE request.order AT in.link.2:
    PLACE status.order AT out.link.2:
    PLACE toOMS        AT in.link.3:
    PLACE fromOMS      AT out.link.3:
    OMS(status , request , toOMS, fromOMS, request.order,
           status.order , accept.data , display.data )
  PROCESSOR 3 T8
    PLACE display.data AT in.link.0:
    PLACE accept.data  AT out.link.0:
    PLACE keyboard     AT in.link.1:
    PLACE screen       AT out.link.1:
    PLACE status.order AT in.link.2:
    PLACE request.order AT out.link.2:
    EMS(screen , keyboard , request.order , display.data , accept.data)

```

5.2.1. Hardware Channel Functions :

The communication among the transputers is performed through the hardware channels. Each channel provides a single route for communication. The SMS module transputer (T1) is connected to the host computer through a channel pair "toHost" and "fromHost". One is used to send data from the transputers to the host via the "toHost" channel. The other is used to get data from the host via the "fromHost" channel. These two channels have a type protocol "ANY". This type protocol is defined by the Occam library. Both the "toHost" channel, and the "fromHost" channel are multiplexed and demultiplexed using software multiplexer and demultiplexer inside SMS module. Fig.5.2.

shows the block diagram of this multiplexer and demultiplexer ² :

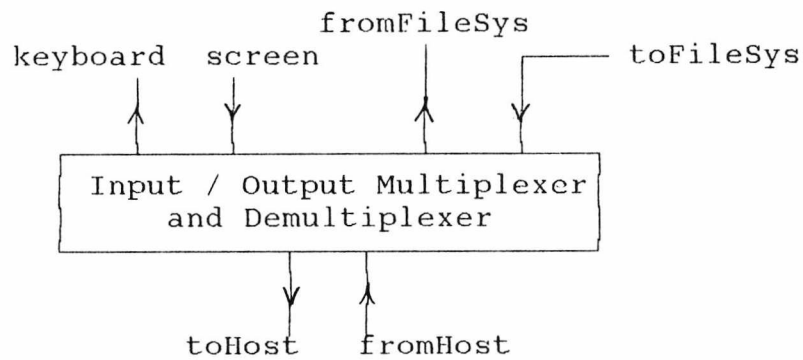


Fig.5.2. Input/Output Multiplexer and Demultiplexer

As shown, the two channels "toHost" and "fromHost" are multiplexed and demultiplexed into :

- i. Two channels "toFileSys" and "fromFileSys". These two channels are the link between the SMS module and the external filing system on the secondary storage database file.
- ii. The "screen" channel which is used to output data from the EMS module to the external user via the "toHost" channel.
- iii. The "keyboard" channel which inputs data from the external user to the EMS module via the "fromHost" channel.

Once the end user sends a message to the EMS module through the "keyboard" channel, the EMS module interprets the user's message. Then, a corresponding system message will be "mailed" to the OMS module (T2) via the "request.order" channel. The "request.order" channel has type "REQUEST.ORDER" (see appendix D.1.). This type protocol is defined by the PKBZ system. It has different type protocols according to the system messages available. It has to be mentioned that any name consisting of lowercase letters combined sometimes with the capital letters represents a channel name, while capital letters alone represents type protocol system definition.

² This process is designed at the University of Kent to facilitate the file handling

When the OMS module (T2) receives the message, the status of the message is returned through the "status.order" channel to the EMS module (T3). Each status value represents a special meaning according to the message.

The transfer of data between the EMS module and the OMS module is achieved via the "display.data" channel and the "accept.data" channel.

The communication between the OMS model and SMS module is performed in similar manner. The system message is sent through the "request" channel, then the status of the message is returned via the "status" channel. The communication of the data between the OMS module and the SMS module is performed through the "toOMS" channel and the "fromOMS" channel. So, the "request" channel has the type protocol "REQUEST.ORDER", and the "status" channel has the type protocol "INT". But, both the "toOMS" channel, and the "fromOMS" have the type protocol "REQUEST.ORDER". This type protocol is defined by the PKBZ system.

** Disadvantage :*

1. The main disadvantage of this system architecture is that the filing system, screen, and keyboard are distributed through the multiplexer and demultiplexer software channels to the host computer. Such design may reduce the system performance. But, there is no other way for communications between the transputers and the external user except through the "toHost" and the "fromHost" channels.

5.3. PKBZ Object Oriented Database Operation :

When PKBZ starts execution, the three basic modules are executing in parallel. Then, the operation in each of the three basic system modules can be illustrated as follows :

i. The EMS module :

While implementing PKBZ, a deadlock situation arose. This occurred while an EMS process was communicating to the screen channel, the keyboard tried to send information, in parallel, to the EMS. In order to overcome such deadlock two new

processes were introduced. These processes are "scr" process, and "key" process as shown in Fig.5.3. :

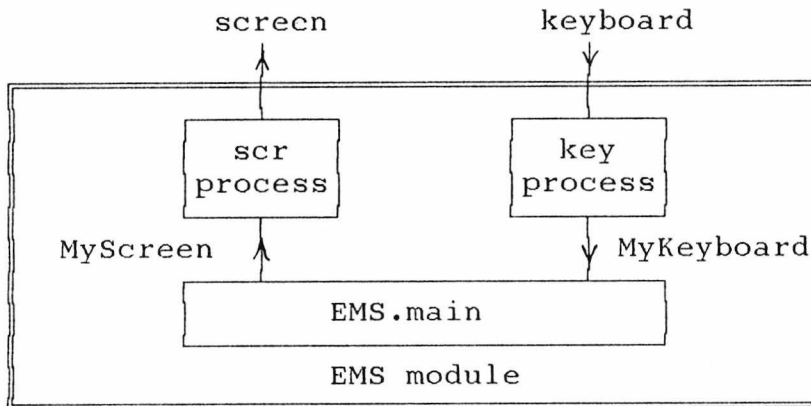


Fig.5.3. Communication Among Processes to Avoid Deadlock

These processes can be considered as two buffers between the other processes in the EMS module and both the screen and the keyboard. Then, the EMS main process can be defined as follows :

```

PROC EMS(CHAN OF ANY screen, keyboard, ... , ..., ... )
  CHAN OF ANY MyScreen , MyKeyboard :
  PAR
    EMS.main(CHAN OF ANY MyScreen , MyKeyboard , ... , ... )
    scr(screen , MyScreen)
    key(keyboard , MyKeyboard)
  :

```

Then, the two processes "scr", and "key" can be defined as follows :

```

PROC scr(CHAN OF ANY screen , MyScreen)
  WHILE TRUE
    INT ch:
    SEQ
      MyScreen ? ch
      screen ! ch
  :

```

```

PROC key(CHAN OF ANY keyboard , MyKeyboard)
  WHILE TRUE
    INT ch:
    SEQ
      keyboard ? ch
      MyKeyboard ! ch
  :

```

By this mechanism, the two above processes are always ready. Then, the deadlock vanishes. So, the EMS.main process can be described as follows :

```

PROC EMS.main (... , ... , ... , ... )
  ... Variables declaration
  SEQ
  continue := TRUE
  WHILE continue
    SEQ
    ... Display screen-1 (see appendix G.1).
    ... Accept user message
    IF
      ... Create Database Message
      ... Perform Create Database Message procedure
      --- (see A in subsection 5.4.1.)
      ... Open Database Message
      ... Perform Open Database Message procedure
      --- (see B in subsection 5.4.1.)
      ... End session
      ... Perform Close Database Message procedure
      --- (see C in subsection 5.4.1.)
      ... TRUE
      ... Send error message to the end user
  :

```

The EMS process accepts only the valid messages from the end user, and invokes the corresponding procedure according to the user's message. The valid message is sent in turn through the "request.order" channel to the OMS module. The complete processes will be described in the next subsections.

ii. The OMS module :

When the system starts operation, the main process in the OMS module can be described as follows :

```

PROC OMS(... , ... , ... , ...)
  ... Variables declaration
  SEQ
  continue := TRUE
  WHILE continue
    SEQ
    request.order ? CASE
      ... Create Dbase
      ... Invoke Create.Dbase Process (see A in subsection 5.4.1.)
      ... Open Dbase
      ... Invoke Open.Dbase Process (see B in subsection 5.4.1.)
      ... END
      ... Invoke Close Process (see C in subsection 5.4.1.)
  :

```

Similarly, the OMS module accepts the user's message from the EMS module, then sends it in turn to the SMS module, and invokes the corresponding process.

It has to be mentioned that deadlock cannot occur in the channels between the OMS module and the SMS module, since the system is designed not to send and receive data at the same time. So, no buffer processes are created.

iii. The SMS module :

Similarly, when the system starts operation, the main process in the SMS module can be described as follows :

```
PROC SMS(... , ... ,           , ... )
  PAR
    Input.Output.Multiplexer.Demultiplexer( ... , ... ,           , ... )
    main.SMS(... , ... ,           , ... )
  :
```

The "Input.Output.Multiplexer.Demultiplexer" process is described in details in subsection 5.2.1. Then, the "main.SMS" process can be described as follows :

```
PROC main.SMS(... , ... ,           , ...)
  ... Variables declaration
  SEQ
  continue := TRUE
  WHILE continue
    SEQ
      request      ?      CASE
        ... Create Dbase
        ... Invoke Create.Dbase Process (see A in subsection 5.4.1.)
        ... Open      Dbase
        ... Invoke Open.Dbase  Process (see B in subsection 5.4.1.)
        ... END
        ... Invoke Close      Process (see C in subsection 5.4.1.)
  :
```

5.4. PKBZ Object Oriented Database Messages :

Fig.5.4 shows the graphical representation of the PKBZ messages. System Messages will be described in subsection 5.4.1 and Internal Messages will be illustrated in subsection 5.4.2. Then, Memory Object Messages will be described in subsection 5.4.3. Finally, Page Object Messages will be illustrated in subsection 5.4.4. Moreover, appendix E. shows the main functions and the data structure of all messages in the PKBZ.

5.4.1. System Messages :

A system message is the message which does not address a specified object class. But, in order to be objective, it is assumed that a virtual object class exists, named

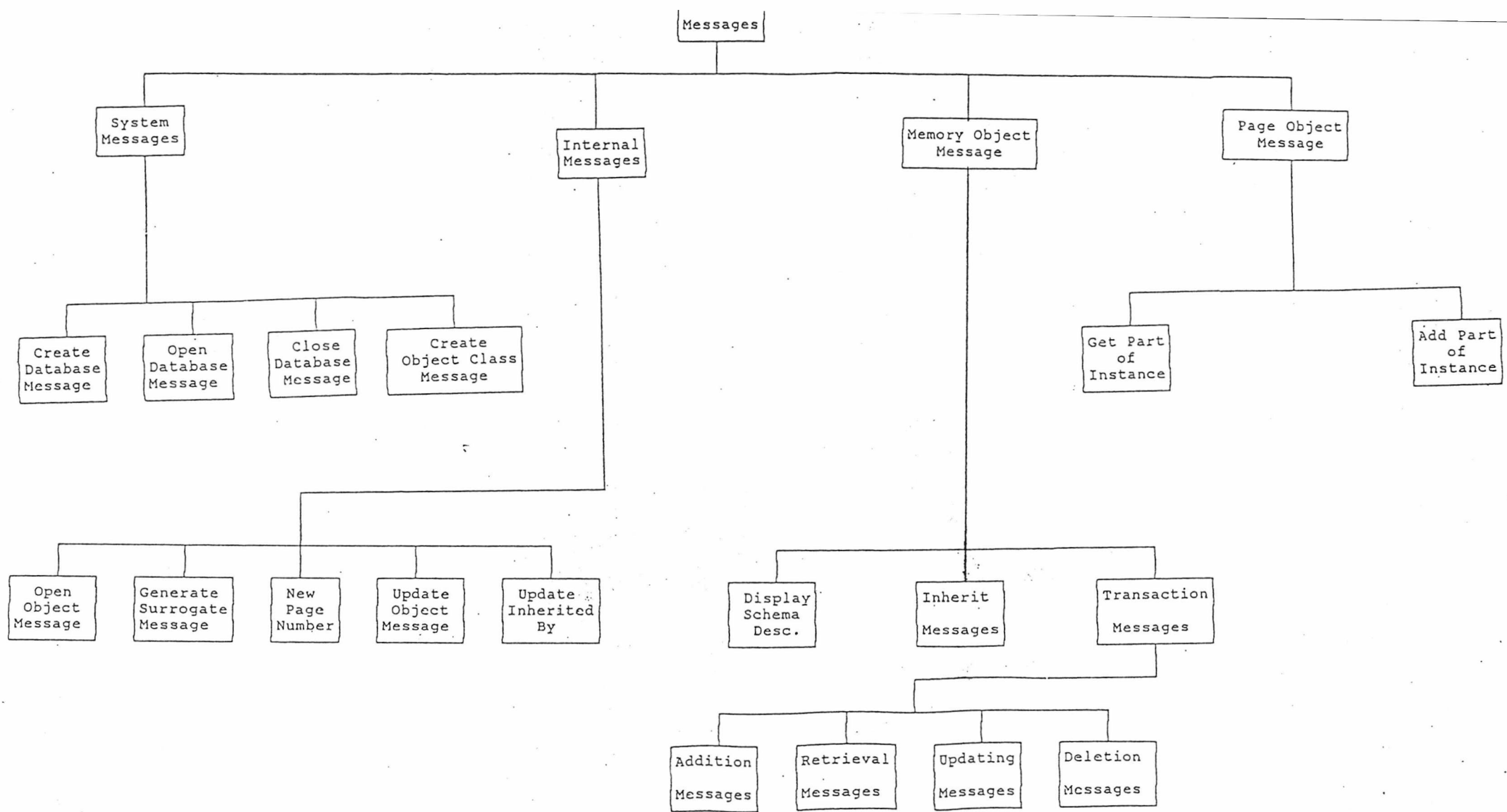


Fig.5.4. Graphical Representation of The PKBZ Messages

the "KBZ.OBJECT". When a system message is sent, the system mails the message to the "KBZ.OBJECT" which in turn processes the corresponding method. The system messages can be classified as shown in Fig.5.4. into :

- a. Create Database Message
- b. Open Database Message
- c. Close Database Message
- d. Create Object Class Message

In the following subsections, each system message will be discussed in detail.

A. Create Database Message :

The database creation message is used to create a new PKBZ OODBMS. Each of the three modules of the PKBZ OODBMS has task to perform this message as follows :

i. The EMS Module :

The following code summarized the process :

```
SEQ
... Accept the database name to be created
... Send CREATE.DBASE message to the OMS module
... Accept the status of the create database in the "stat" variable
IF
    stat <> 0    -- Database is not created
        ... Interprets the status meaning to end user
    TRUE  -- Database is created
        ... Send a message to the end user that the database is created
... Display screen (screen-1) (see Fig.G.1)
```

If the status has non zero value, this means that an error is encountered during the creation process. For example, an old PKBZ OODBMS with the same name already exists.

ii. The OMS module :

When the OMS receives the message, it invokes the "Create.Dbase" process. The "Create.Dbase" process can be summarized as follows:

```
PROC Create.Dbase ( ... , ... , ... )
... Variables declaration
SEQ
... Send the create database message to the SMS module
... Accept the status of the creation process
```

... Send the status to the EMS module

:

iii. The SMS module :

The SMS module performs the main task of this message. The SMS module invokes an internal process called "Create.Dbase" process. It has the same name as in the OMS module.

The main basic function of the "Create.Dbase" process is to create the system object classes (see subsection 4.3.) of the new database and to store them into the new database file. The general block diagram of the "Create.Dbase" process is shown in Fig.5.5.

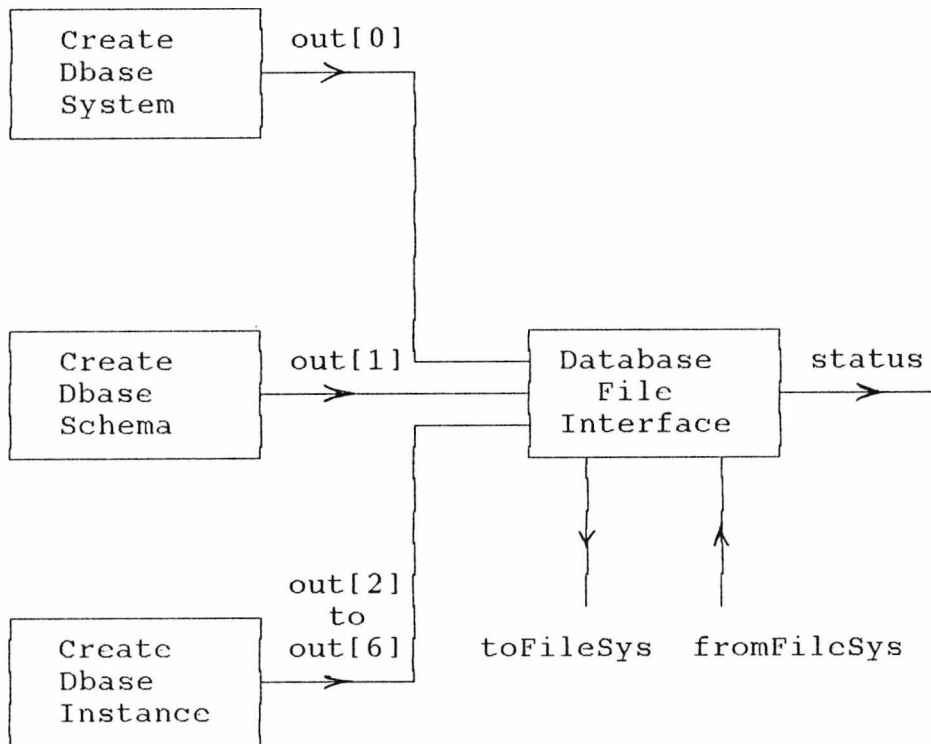


Fig.5.5. Create.Dbase Process in The SMS Module

The Create.Dbase process can be described as follows :

```
PROC Create.Dbase ( ... , ... , ... )
  ... Variables declaration
  PAR
    ... Invoke Create.Dbase.System      Process
    ... Invoke Create.Dbase.Schema      Process
```

```
... Invoke Create.Dbase.Instance    Process
... Invoke Database.File.Interface Process
```

:

When the database file name is passed to the "Database.File.Interface" process, then the process checks the existence of the database file in the system disk. If there is no file of the same name, the process will send a zero status through the "status" channel to inform the OMS module that the database is created, although nothing is created yet. But, the system uses the Complete/Incomplete Parallel Processing technique which is described in chapter 3.

The "Create.Dbase.System" process stores the initial values of the data corresponding to the "Database System Group" of the database file (see Fig.4.9.). That is, the "sys.info" values which are described in subsection 4.5.3. Such values are transformed into a page format, as described before. Then, the corresponding page is passed to the channel "out[0]".

At the same time, the "Create.Dbase.Schema" process is processing in parallel too. The schema definition of all the five system object classes are stored in this process. These definitions are transformed into a single page corresponding to the Schema System Group of the database file. Then, this page is sent to the channel "out[1]".

Similarly, the "Create.Dbase.Instance" process stores the initial values of all object instances corresponding to system object classes. Each class of object instances is transformed into a single page. Then, each page is passed to the corresponding one of the last five channels of the "out" channels. That is, out[2] to out[6]. These pages correspond to the Instance System Group of the database file.

When a page is received, the "Database.File.Interface" process writes this page into the corresponding page in the secondary storage database file through the "toFileSys" channel. The status of writing is responded through the "fromFileSys" channel.

It has to be mentioned that while the "Database.File.Interface" is checking the database file existence in the secondary storage, the other blocks are processing in parallel too. Moreover, using the Complete/Incomplete Parallel Processing technique, the

system objects are created in the background.

B. Open Database Message :

The main function of the "OPEN.DBASE" message is to load the system object classes together with the necessary processes into the main memory. Then, when these processes are loaded, the system is ready to accept any further messages. The operation of this message in different modules can be described as follows :

i. The EMS module :

The main function of the open database message can be summarized as follows :

```
SEQ
... Accept the database name to be opened
... Send OPEN.DBASE message to the OMS module
... Accept the status of the open database in the "stat" variable
IF
  stat <> 0    -- Database is not opened
    SEQ
      ... Interprets the status meaning to end user
      ... Display screen (screen-1) (see Fig.G.1.)
  TRUE -- Database is opened
    SEQ
      ... Send a message to the end user that the database is opened
      ... Display screen (screen-2) (see Fig.G.2.)
```

ii. The OMS module :

When the "OPEN.DBASE" message is received, the "Open.Dbase" process is invoked.

The "Open.Dbase" can be summarized as follows :

```
PROC Open.Dbase ( ... , ... , ... )
... Auxiliary Variables Declaration
SEQ
... Send the OPEN.DBASE message to the SMS module
-- to open database file through "request" channel
... Accept the status of the open database from the SMS module
... Send the status of the opened database to the EMS module
IF
  stat <> 0 -- Database is not opened
    SKIP
  TRUE -- Database is opened
    PAR -- (See Fig.5.6. OMS module) Invokes the following processes
      ... Stop.Process
      ... Schema.Instance.Distribution
      ... Object.Memory.Management
      ... Object.Manager
```


- ... All System Memory Objects
- ... Other Memory Objects are waiting to be created

:

If the database file exists, the OMS module will send a zero status to the EMS module before the process is totally completed. Then, the rest of the process will be completed in the system background.

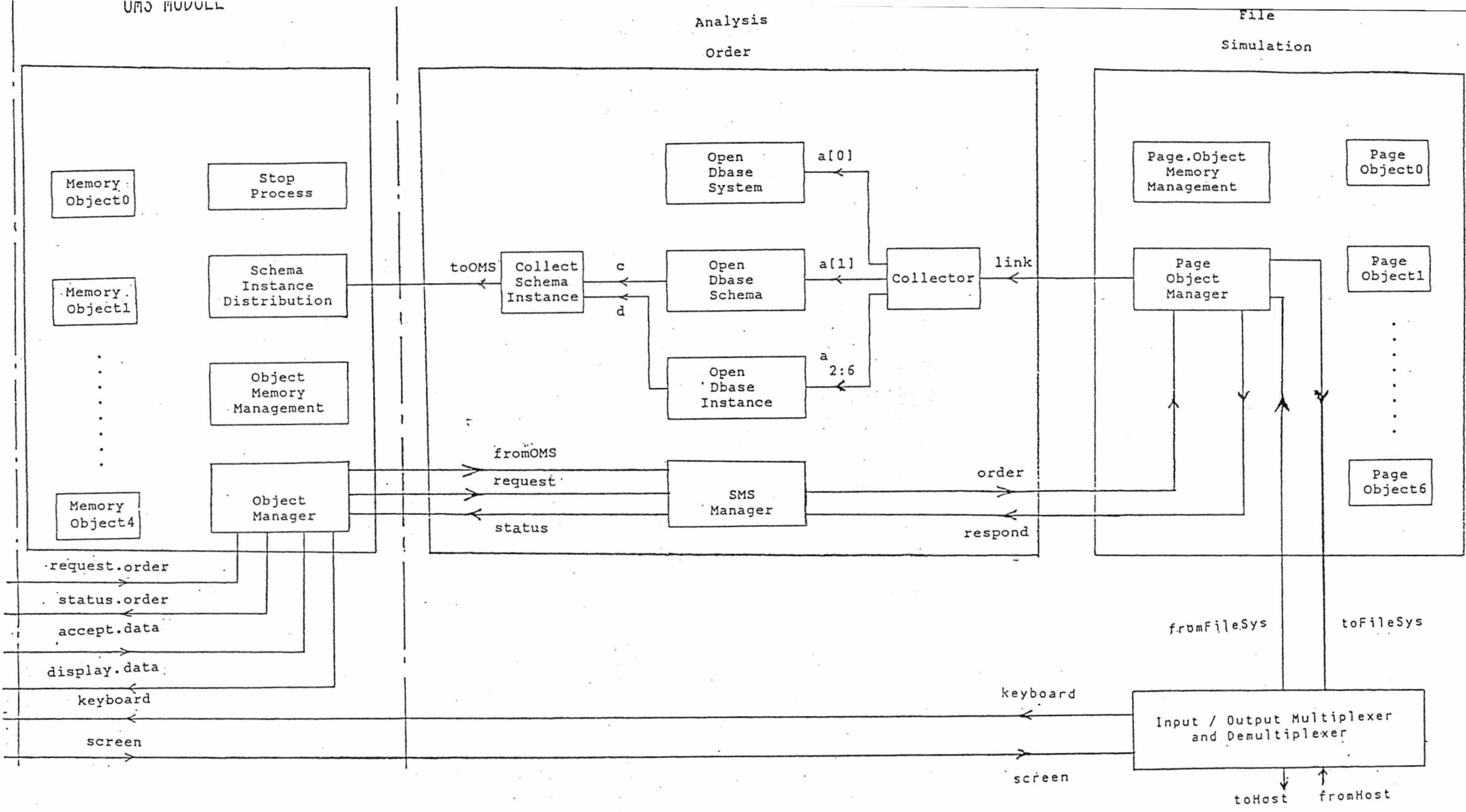
When the database is opened, all the system object classes information are buffered from the SMS module to the "Schema.Instance.Distribution" process via the "toOMS" channel in sequence. Then, each system Memory Object process receives the corresponding object schema and object instances via the "in.schema" and the "in.instance" channels respectively (see Fig.4.1.).

When the "Schema.Instance.Distribution" process receives the schema and instance information, it sends the following information to the "Object.Memory.Management" process :

- Object identifier
- Object name
- Memory Object location in the system

Hence, the "Object.Memory.Management" process keeps track of the location of each Memory Object. During the system operation, the system consults this process to get the corresponding Memory Object position in the system. The "Object.Manager" process is responsible for controlling the messages between the Memory Objects and the external world. That is, when any further message is received, it is directed to the "Object.Manager" process, then the "Object.Manager" process sends the message to the corresponding Memory Object in the system.

After the database is opened, only five Memory Objects are invoked corresponding to the system object classes. Any further Memory Object is created upon "Object.Manager" process request. That is, the number of Memory Objects is variable. It depends upon the number of object classes needed while the system is active. A similar dynamic process will be described in the SMS module within the File Simulation process in the SMS module.



Open Process Block Diagram
in The OMS Module

Open Process Block Diagram
in The SMS Module

Fig.5.6. Block Diagram of the Open Database Message in OMS and SMS modules

It has to be mentioned that the interconnection channels among processes in the OMS module are not shown to simplify the diagram.

iii. The SMS module :

When the SMS module receives the "OPEN.DBASE" message, the corresponding "Open.Dbbase" process is invoked. The "Open.Dbbase" process can be defined as follows :

```
PROC Open.Dbbase ( ... , ... , ... , ... )
  ... Auxiliary Variables Declaration
  PAR — Invoke the following process (see Fig. 5.7.)
    ... File.Simulation Process
    ... Analysis.Order Process
:
```

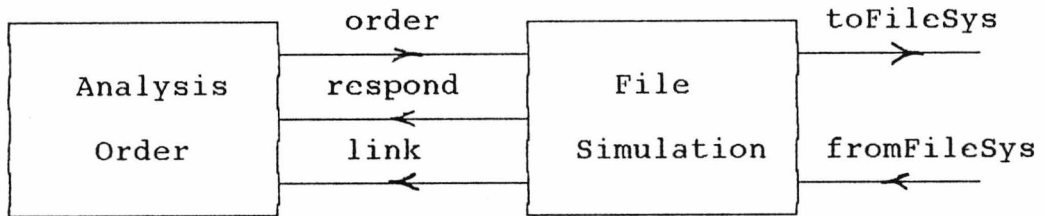


Fig.5.7. Open Process Block Diagram in The SMS Module

The "File.Simulation" process is used as a buffer between the filing system and the rest of the processes in the SMS module. The order to the "File.Simulation" process is sent through the "order" channel. Then, the status of the order is returned through the "respond" channel. The "link" channel is used for data transfer between the "File.Simulation" process and the "Analysis.Order" process.

* File Simulation Process :

The following code represents the main functions of the "File.Simulation" process:

```
PROC File.Simulation.Process ( ... , ... , ... , ... )
  ... Auxiliary Variables Declaration
  SEQ
    ... Check the Database File
    ... Send the status to the Analysis.Order Process
  CASE stat
    0 — Database file exists
      PAR — Invokes the following processes (see Fig.5.6.)
        ... Page.Object.Manager
```

```

... Page.Object.Memory.Management
... Page.Object processes are waiting to be created
--- upon Page.Object.Manager request
ELSE
  SKIP
:

```

If the database file exists, the "Page.Object.Manager" process starts execution, and reads the following pages from the secondary storage database file (see Fig.4.9.):

- a. Database System Group Page (i.e. Page zero)
- b. Schema System Group Page (i.e. Page one)
- c. Instance System Group Pages corresponding to the system object classes instances (i.e. Pages two to six)

The main function of the File Simulation Process is that when a new page is read, a new "Page.Object" process is created in the system, and the content of the page from the secondary storage database file is buffered, into this "Page.Object" process in the main memory. Then, any further retrieval or updating will be performed from the main memory for fast action, as described in chapter 3.

Since, it cannot be known in advance how many pages are needed during the run time, the number of "Page.Object" processes has to be a variable number declaration. But, Occam does not allow a variable number declaration. In order to overcome this problem and the same problem in the OMS module, a special mechanism is carried out. The corresponding Occam code in the File Simulation process Fig.5.6. is as follows :

```

[max.number.page.object]CHAN OF BOOL new:
[max.number.page.object]BOOL continue:
PAR
  Page.Object.Manager(... , ... , new , ... )
  PAR i = 0 FOR max.number.of.page.object
    SEQ
      new[i] ? continue[i] -- Wait for signal
      IF
        continue[i] = TRUE -- Create new Page Object
          PAR
            Object.Schema( ... , ... , ... )
            Page.Object( ... , ... , ... )
          TRUE -- Page.Object will not be created
          SKIP
  Page.Object.Memory.Management( ... , ... , ... )

```

The Occam code specifies that the following processes "Page.Object.Manager" is

processing in parallel with the "Page.Object.Memory.Management". But, the "Page.Object" is not created until a signal from the "Page.Object.Manager" is sent to the "new[i]" channel. If the "continue[i]" is true, a new "Page.Object" process will be created. By this mechanism the system keeps track and generates only the "Page.Object" process according to the system requirements. The "Object.Schema" process has the schema definition. This definition is sent to the "Page.Object" via "in.schema" (see Fig.4.1.) channel to the corresponding "Page.Object", when the "Page.Object" is created.

The "Page.Object.Memory.Management" process stores and keeps track of all pages which are read from the secondary storage. During the system life, this process is consulted to prevent a single page being read twice.

* *Analysis Order Process :*

The following code represents the main functions of the "Analysis.Order" process:

```

PROC Analysis.Order ( ... , ... , ... , ...)
  ... Auxiliary Variables Declaration
  SEQ
    ... Accept the opened database file status from file simulation process
    ... Send the status to the OMS module
  CASE  stat
    0  -- Database file exists
      SEQ
        PAR
          ... Send an order to the File.Simulation.Process
          --- to read the system objects
          -- Invokes the following processes (see Fig.5.6.)
          ... Collector
          ... Open.Dbase.System
          ... Open.Dbase.Schema
          ... Open.Dbase.Instance
          ... Collect.Schema.Instance
        continue := TRUE
      WHILE  continue
        ... Accept Message from OMS Module and Invokes the Corresponding Method
    ELSE  -- Database file is not opened
      SKIP
  :

```

It should be noted that in the Analysis.Order process, if the corresponding database file exists, the zero status will be sent to the OMS module through the "status" channel and the rest of the operation will be carried out using the Complete/Incomplete Parallel Processing technique.

Each page read is buffered to the "Analysis.Order" process through the "link" channel. Then, the "Analysis.Order" process transforms the pages from the physical data representation into the logical object classes representation.

The "Collector" process acts as a buffer between the "link" channel and the other processes in the "Analysis.Order". Moreover, it distributes each received page into the corresponding process. For example, when the first page in the database file is sent to the "Collector" process, it buffers the page to the "Open.Dbase.System" process. Since the first page is corresponding to the Database System Group (see subsection 4.5.), then the "Open.Dbase.System" process converts this page from the physical data representation to the logical data representation in the form of a single integer array (see appendix F). This integer array is consulted and updated during the system life.

When the second page (Schema System Group), in the secondary storage database file, is sent to the "Collector" process, the "Collector" buffers the page to the "Open.Dbase.Schema" process. This process transfers the physical system object classes schemes (Fig.4.10.) to the logical representation. Then, each object schema is buffered to the "Collect.Schema.Instance" process.

Similarly, the system object page instances (i.e. pages 2 to 6) are buffered in sequence to the "Open.Dbase.Instance" process. This process transfers the physical system object instances representation to the logical object instances representation. Then, the instances of each object class are buffered to the "Collect.Schema.Instance" process.

When both object schema and object class instances are buffered to the "Collect.Schema.Instance" process through the "c" and "d" channels respectively, then this data is sent to the "Schema.Instance.Distribution" in OMS module via the "toOMS" channel. In turn, all system object classes will be buffered into the corresponding Memory Objects in the OMS module. Then, the "SMS.Manager" process is ready to accept any further message.

It has to be mentioned that the communication channels among processes in the SMS module have not been shown in order to simplify the diagram. Moreover, all the previous processes in the Fig.5.6. are processing in parallel in both the OMS module, and

the SMS module when executing the open database message. Each process executes a certain task concurrently. This with no doubt, reduces the response time. Moreover, using the Complete/Incomplete Parallel Processing, the response to the "Open.Dbase" message is reduced too. That is because while the screen-2 is displaying and the end user is typing the next message, the system objects are opened in parallel.

C. Close Database Message :

The main function of the close database (END message) is to insure that all the transactions which have occurred while the system has been active are updated into the secondary storage database file. The updating transaction is carried out in two stages. The first stage updates the relevant Page Object(s) in the SMS module according to the transaction in the Memory Object(s) in the OMS module. Then, all Page Objects are updated to the secondary storage database file. The complete procedures in each module will be as follows :

i. The EMS module :

The main function of the EMS module can be summarized as follows :

```
SEQ
  ... Send END message to the OMS module
  ... Accept the status of the message
  ... Display the status to the end user
```

ii. The OMS module :

When the OMS module receives the END message, the "write.update.objects" process is invoked. The process can be summarized as follows :

```
PROC write.update.objects(... , ... , ... , ... )
  ... Variables Declaration
  SEQ
    SEQ i = 0  FOR  open.object.now
      SEQ
        ... Send UPDATE.INSTANCE message to Memory.Object[i]
        --- (see D. in subsection 5.4.2.)
        ... Accept status from Memory.Object[i]
      CASE  stat
```

```

none -- Not updated
SKIP
ELSE -- Memory Object is update
SEQ
... Accept the instances
... Convert instances into page
... Send the page to the SMS module
... Accept the object schema
... Convert schema into string
... Send the string to the SMS module
... Send END message to Memory Object to stop processing
... Send END to the SMS module
... Accept the status from SMS module
... Send the status to the EMS module
:

```

It has to be mentioned that the previous code can be processed using Complete/Incomplete Parallel Processing technique. But, in the unlikely event that an error occurs, then the end user may receive an inappropriate message. So, this message especially does not use the Complete/Incomplete Parallel Processing technique.

iii. The SMS module :

When the SMS module receives the END message, then the "write.update.Page.Object" process is invoked. The process can be summarized as follows:

```

PROC write.update.Page.Object( ... , ... ,      , ... )
... Variable Declaration
SEQ
... Check each updated Page Object
... Write the updated instances to the secondary storage
... Send END message to the Page Object to stop
... Send the status to the OMS module
:

```

Then, all the other processes stop executing.

D. Create Object Class :

Once the database is opened, the end user is able to create new object classes. The current prototype allows the end user to create any KBZ object class of the following types : Basic Object, Structural Object, and Entity Aggregate Object. The following subsections describe the creation procedures :

i. The EMS module :

When the screen-2 (Fig.G.2.) is displayed and the end user chooses "KBZ.CreateObject" message, then the "Create.Object" process is invoked. This process can be described as follows :

```
PROC Create.Object( ... , ... , ... )
  ... Variable Declaration
  SEQ
  ... Display Screen-3 (Fig.G.3.)
  ... Accept object class type
  CASE object.type
    entity.set
      ... Invoke Create.Entity.Set           process
    attribute.set
      ... Invoke Create.Attribute.Set       process
    structural.object
      ... Invoke Create.Structural.Object   process
    entity.aggregate
      ... Invoke Create.Entity.Aggregate.Object process
  :
```

All the previous creation processes are implemented. As an example, the "Create.Structural.Object" will be described only.

```
PROC Create.Structural.Object ( ... , ... , ... , ... )
  ... Variables Declaration
  SEQ
  ... Display screen-4 (Fig.G.4.)
  ... Accept object name
  ... Accept property names and property types (see B.2. in subsection 4.2.1.)
  ... Send object name, and property types to the OMS module to perform
  --- validation check
  ... Send a message to the OMS module to create new object class, if the checks
  --- are valid
  ... Accept the status from OMS and display it to the end user
  :
```

In general, some necessary validation checks are carried out during the data entry. These checks are carried out by sending different messages to the OMS module. For example :

- a. Object class name has to be unique
- b. If the object class is a basic object, then the object instance type will be either integer or string.
- c. If the object class is a structural object, then both the domain and the range

of the relationship must be previously defined. Moreover, the domain of the primary relationship must be an entity set.

- d. If the object class is an entity aggregate object, the inherits objects must be structural objects.

The previous validations are essential for data consistency. So, the system prevents the end user from entering any invalid data.

ii. The OMS module :

When the OMS module receives the create object message, then the "Create.Object" process is invoked. This process can be described as follows :

```
PROC Create.Object( ... , ... , ... )
  ... Variable Declaration
  SEQ
    ... Accept object schema
    ... Send a message to the "OBJECT-ID" entity object to generate new surrogate
    --- (i.e. object identifier of the new object class)
    ... Send a status to the EMS module that object is created
  PAR
    ... Send a message to the SMS module to create new object class
  SEQ
    ... Accept the position of the object schema in the Schema System Group
    --- page (see Fig.4.10.) from SMS module
    ... Accept the position of the corresponding page instance in the
    --- Instance System Group from SMS module
    ... Accept the status of the creation from the SMS module
    ... Add the new object class information in all the System Object classes
    --- (see subsection 4.3.)
  SEQ
    ... Create new Memory Object in the OMS module
    ... Send the object information to the corresponding Memory Object
  SEQ
    ... Update the inherited by information in all inherit object classes by
    --- the new object class, if any (see subsection E. in 5.4.2.)
    ... Send object class information to the "Object.Memory.Management" process
    ... Send object class information to the "Schema.Instance.Distribution"
    --- process
  :
```

iii. The SMS module :

When the "SMS.Manager" process accepts the create object message, the following code describes the process in "SMS.Manager" as follows :

```

PAR
  SEQ
    ... Send the position of the object schema in the Schema System Group page
    --- to OMS module
    ... Send the next available page in the Instance System Group
    --- to the OMS module (i.e. sys.info[0], see subsection 4.5.3.)
    ... Send the status to the OMS
  SEQ
    ... Transform the new object schema into string (see subsection 4.4.2.)
    ... Send the string to the Schema System Group
  SEQ
    ... Send a message to the File Simulation process to create new Page Object
    ... Update System Information "sys.info"
    ... Send a message to the File Simulation process to update Database System
    --- Group (i.e. sys.info array)

```

As shown, the creation process is sophisticated process.

5.4.2. Internal Messages :

Internal messages are messages which are executed internally by the system only.

The internal messages can be classified as shown in Fig.5.4. into :

- a. Open Object Message
- b. Generate Surrogate Message
- c. New Page Number Message
- d. Update Object Message
- e. Update Inherited By Message

With the System Messages, the detail of the processes in each module were illustrated, but in the following subsections, the function of the internal message will be described only together with the parallelism aspects.

A. Open Object Message :

The main function of this message "OPEN.OBJECT" is to transform the physical object representation from the database file into the logical object representation in the form of a Memory Object. That is, to build the corresponding Memory Object in the main memory. Once the Memory Object is in the memory, it will be ready to accept any other object messages.

It has to be mentioned that the Complete/Incomplete Parallel Processing technique is performed with this message, since the end user is informed that the corresponding object class is loaded, while the SMS module transforms the physical object representation into a logical representation. Moreover, a new Page Object is allocated in the File Simulation process, and the OMS allocates a new Memory Object.

B. Generate Surrogate Message :

This message "GENERATE.SURROGATE" is accepted only by the entity set type. It is issued by the system to generate a new surrogate when a new object instance is added. The surrogates are generated sequentially starting from zero. The "instance.description[4]" (see E in subsection 4.2.2.) keeps tracks of the surrogate values in the object class schema.

It has to be mentioned that the updating of the instance description to the secondary storage is performed in the system background.

C. New Page Number Message :

This message "NEW.PAGE.NO" is used in PKBZ version-2. It generates a new page in the secondary storage database file, when a new instance is added and the current page is not able to accommodate the new instance.

The message is performed using Complete/Incomplete Parallel Processing, since a new Page Object is allocated in the SMS in the system background, during execution of the add instance process.

D. Update Object Message :

This message "UPDATE.INSTANCE" is used in both the OMS module and the SMS module. In the OMS module, it reflects any transaction from the Memory Object to the corresponding Page Object using Object Update Parallel Processing. In the SMS module, it transfers the updated Page Object instances from the File Simulation Process to the secondary storage database file. The message is carried out in the system background while the system is being active.

E. Update Inherited By Message :

"UPDATE.INHERITED.BY.OBJECT" is an internal message. The system uses this message when a new object class is created to update the corresponding inherited by information by the related created object class, if any. This process is carried out as a part of the Create Object Message in the OMS module. But, the corresponding method inside each relevant Memory Object is invoked for updating inherited by object class information inside this Memory Object.

5.4.3. Memory Object Message :

A Memory Object Message is a message which addresses a specific object class and is available to the external user. Such message is directed to the corresponding Memory Object in the OMS module by the Object Manager process (see Fig.5.6.). Then, the corresponding method is invoked inside the Memory Object. The Memory Object message can be classified, in general as shown in Fig.5.4. into :

- a. Display Schema Description Message
- b. Inherit Messages
- c. Transaction Messages :
 - * Addition Messages
 - * Retrieval Messages
 - * Updating Messages
 - * Deletion Messages

The following subsections describe only the function of each message.

A. Display Schema Description Message :

The Display Schema Description message is sent to any object class to get the following schema information :

1. Object name
2. Property names and types
3. Inherits object names (if any)

4. Inherited by object names (if any)

When the end user chooses "SchemaDesc" in screen-2 (Fig.G.2.), the system processes internally the "OBJECT.NAME.SCHEMA.DESCRPTION" message. Then, the previous information are displayed on the screen. The EMS module displays the object schema according to the object class type. For example, if the object class is a basic type, the property name with property type will be displayed. But, if the object class is a structural object, both the primary and the inverse relationships together with the domain and range are displayed. Further, if the object class is an entity aggregate object, the corresponding relationships names and their types are displayed. That is, the EMS module chooses automatically the proper screen shape according to the object class type.

B. Inherits Messages :

The inherit object classes identifiers and the inherited by object classes identifiers are stored together with any object class. The retrieval of this information with the corresponding class names is performed within the Display Schema Description message.

C. Transaction Messages :

These messages are corresponding to the traditional transactions in any database system. They are used to add, retrieve, update, and delete instance(s) in any object class.

C.1. Addition Messages :

The addition messages are used to add a new object instance to the different object class types. The addition messages can be classified into :

1. Add an instance to the basic object (Integer, String)
2. Add an instance to the structural object
3. Add an instance to the entity aggregate object

The function of the addition processes will be described as follows :

When the screen-2 (Fig.G.2.) is displayed, the end user specifies the object name in which an instance has to be added and then specifies "AddInstance" in the message

zone. So, the system allows the end user to add an instance in the corresponding object class. This message is called "ADD.INSTANCE". The system is intelligent enough to provide the end user with the corresponding suitable screen to add the new instance. The following validations are carried out during adding processes :

- a. An instance cannot be added to the entity set
- b. Any object instance value must satisfy the object instance definition in the schema according to the object type.

It has to be mentioned that the object instances are added in sequence in chronological order. For simplicity, no indexed mechanism is carried out during the addition process in the prototype. It is accepted that the search is slow with large volumes of data. But, in future implementations, an index mechanism could be carried out for fast retrieval.

The Memory Object uses the "overloading" mechanism to choose the appropriate method according to the object type.

In the case of the entity aggregate object, the addition process is carried out in parallel, since all instances are added in the corresponding inherit structural objects simultaneously.

It has to be mentioned that the corresponding updating to the secondary storage database file is performed in the system background using two techniques : Object Update Parallel Processing and File Simulation Processing.

C.2. Retrieval Messages :

The main function of this message is to get the instance values in different object types. There are various types of object retrieval messages. The following subsections explain the difference.

** Check Instance Existence :*

The function of this message "InstanceVal" (Fig.G.2.) is to check if a certain value of instance already exists in the object instances or not. It is used only with the basic object and the structural object. This message is called "GET.INSTANCE".

** Get All Object Instances :*

The "GetInsValue" (Fig.G.2.) provides a way to retrieve all object instances of any object types. This message is processed internally by "OBJECT.NAME.INSTANCE" message.

The system is intelligent enough to provide the end user with the corresponding suitable screen to display the instances according to the object type.

In the case of the entity aggregate object, the retrieval process is carried out in parallel, since all instances are instantiated from the corresponding inherits structural objects simultaneously.

** Get Specific Values :*

This message function depends upon the object type. For example, in the structural object, if the end user gives the domain of the primary relationship, the system responds with the corresponding range value instance(s) according to the type of relationship (one-to-one, one-to-many). This message is called "GetPriRange" (i.e. Get Primary Range) (see screen-2 Fig.G.2.). It is processed internally by "GET.PRI.RANGES" message. But, if the end user gives the range of the primary relationship, the system will respond with the corresponding domain. This message is called "GetInvRange" (i.e. Get Inverse Range). It is processed internally by "GET.INV.RANGES" message. In addition, in the entity aggregate object, if the end user gives one or more instance value(s) in the range of a structural object related to the entity aggregate object, then the system responds with one or more instances which satisfy the condition(s), if any. This message is called "GetSpeValue" (i.e. Get Specific Value) (see screen-2 Fig.G.2.). It is processed internally by "GET.SPE.VAI.UE" message.

It has to be mentioned that the entity aggregate object instances are instantiated during the run time, in parallel, and the selection is carried out after instantiation to satisfy the selection criteria.

C.3. Updating Messages :

"UpdateIns" (Fig.G.2.) is used to update the object instance(s) in the different

object class types. It is processed internally by "UPDATE.INSTANCE.VALUE" message. The updating messages can be classified into :

1. Update an instance in the structural object
2. Update an instance in the entity aggregate object

The function of the updating processes will be described only as follows :

When the screen-2 (Fig.G.2.) is displayed, the end user specifies the object name in which an instance has to be updated. The system is intelligent enough to provide the end user with the corresponding suitable screen to update the instance according to the object type. Any updated instance value must satisfy the object instance definition in the schema according to the object type.

Moreover, the Memory Object uses the "overloading" mechanism to choose the appropriate method according to the object type.

It has to be mentioned that the corresponding updating mechanism to the secondary storage database file is performed in the system background using two techniques : Object Update Parallel Processing and File Simulation Processing.

C.4. Deletion Messages :

"DeleteIns" (Fig.G.2.) is used to delete the object instance(s) in the different object class types. It is processed internally by "DELETE.INSTANCE.VALUE" message. The deletion messages are executed in entity set, structural object and entity aggregate object. When the screen-2 (Fig.G.2.) is displayed, the end user specifies the object name in which an instance has to be deleted and the system provides the end user with the corresponding suitable screen to delete the instance according to the object type.

It has to be mentioned that the corresponding deletion mechanism to the secondary storage database file is performed in the system background using two techniques : Object Update Parallel Processing and File Simulation Processing.

5.4.4. Page Object Messages :

The system uses these messages to deal with the Page Objects in the SMS module. The Page Object messages can be classified, in general as shown in Fig.5.4. into :

a. Get Part of Instance

b. Add Part of Instance

The following subsections describe each message :

A. Get Part Of Instance :

In the File Simulation Process, the Page Object stores the records in the form of the instances. One record accommodates more than one Memory Object Instance. So, this message "GET.PART.OF.INSTANCE" is used to get a portion of a record only. For example, it is issued internally by the system to get the string corresponding to any object schema from the related Page Object which stores the Schema System Group (see Fig.4.10).

B. Add Part Of Instance :

This message "ADD.PART.OF.INSTANCE" is used to add or update a part of instance in the Page Object. For example, when a new Memory Object is created, the string corresponding to the schema is sent to the relevant Page Object with this message to be added. On the other hand, this message is used to update a part of instance in the Page Object. For example, when an object schema is changed, this message is used to perform the updating in the corresponding Page Object.

5.5. Conclusion :

The processes inside each of the three main modules (EMS, OMS, and SMS modules) together with their functions are illustrated and how these processes communicate with each other.

The system allows the end user to communicate with the object classes through the message sending. The different types of messages and its corresponding method implementations are discussed.

The system creates a variable number of both the Memory Objects and the Page Objects according to the need. Although Occam does not allow a variable number definition, a mechanism is discussed which overcomes such a limitation.

Since, in PKBZ version-1, only one Memory Object representation is implemented for

all different types of the object classes, then a single method name may denote different method implementation according to the object class type. This mechanism is defined as "overloading" process, as described in chapter 1. The PKBZ version-1 supports the "overloading" mechanism by choosing the appropriate implementation according to the object type automatically during the run time. The main advantage of this property is that the end user is not aware of the different codes necessary to implement the method.

In general, the system is of a truly object oriented nature. Everything is in the form of an object class. Processes are executed only by invoking a related message inside the related object class. Moreover, object classes persist until they are destroyed.

It has to be mentioned that all the different aspects of parallel processing which are discussed in the chapter 3 are implemented except Parallel Data Distribution. Moreover, only one type of the three Parallel System Architecture is implemented, since it is not possible to implement all the three types in a single prototype.

CHAPTER 6

PKBZ Version-2 and Experimental Results

6.1. Introduction :

In this chapter the description of the enhancements and modifications to PKBZ version-1 will be described to get PKBZ version-2. The main enhancements and modifications of version-2 are :

1. The number of instances per Memory Object is increased. Also, the object class instances can be stored in more than one page in the secondary storage database file.
2. The EMS module is changed to allow not only on-line processes, but also batch processes.

These two enhancements are necessary in order to carry out experiments with large volumes of data to measure the system performance.

3. The Memory Object data structure is changed so that each KBZ object class type is mapped into a different Memory Object structure.

This modification is designed to reduce the size of each Memory Object by specializing the Memory Objects.

4. The communication mechanism is changed to reduce the number of both the hardware channels among transputers and the software channels inside each transputer.

This modification is designed to reduce the unnecessary communication channels which can be merged with other channels.

The two modifications are discussed first. Then, the enhancements are discussed, and this is followed by a performance evaluation of the system on different numbers of transputers.

6.2. An Object Class Logical Representation (Memory Object)

Version-2 :

As mentioned before, in PKBZ version-1, all the different object classes, including Page Object, are mapped into the same Memory Object structure. But, in PKBZ version-2, each KBZ object class type is mapped into a different Memory Object structure. So, different Memory Object types are designed and implemented according to :

- i. Object class type
- ii. Object class instance type

Table 6.1. illustrates the different Memory Objects which are designed and implemented in PKBZ version-2 :

No.	Object Name	Object Class Type	Instance Type
1	Entity Object	Entity Set	Integer
2	Attribute.Object.Int	Attribute Set	Integer
3	Attribute Object.String	Attribute Set	String
4	Structural Object.Int	Structural Object	Primary Domain is Integer Primary Range is Integer
5	Structural Object.String	Structural Object	Primary domain is Integer Primary range is String
6	Entity Aggregate.Object	Entity Aggregate	Integer
7	Page.Object	Page Object	Record

Table 6.1. Different Memory Objects in PKBZ version-2

The schema structure of all the previous Memory Objects has the same data structure. But, the instance data structure is different. It depends upon the instance type. The main advantages of this splitting are :

1. The system becomes truly object oriented, since each object class type is mapped

into the corresponding Memory Object.

2. The overloading is not performed through the CASE statement as in version-1, since each Memory Object stores the corresponding messages only.
3. Page Object is totally different in both message and instance data structure, so it is better to keep it separate rather than integrate it with the other objects.
4. Consequently, the size of Memory Object becomes smaller than in PKBZ version-1.

But, the main disadvantage is :

1. The system becomes more complicated, since the system has to map each object type to the corresponding Memory Object, during the running process. Thus, both the object class and the type have to be checked before the mapping. But, in version-1, such checks are not carried out.

6.3. Communication Channel Modifications :

In PKBZ version-2, both the hardware and the software channels are modified. The hardware channels are modified by introducing a new PKBZ general block diagram. This modification will be discussed in subsection 6.3.1. On the other hand, the software channels are modified by changing the Memory Object communication mechanism. This modification will be discussed in subsection 6.3.2.

6.3.1. PKBZ version-2 General Block Diagram :

During the implementation phase of version-1, we notice that although the order is passed via "request.order" channel, and the data is sent through "accept.data" (see Fig.5.1.) in parallel, both of them are executed in sequence. Then, it is better to merge the two channels together. So, in version-2, the "accept.data" channel is merged with the "request.order" channel.

Similarly, in version-1, the status of the operation is displayed through the "status.order" channel, followed by any output data, if any, through the "display.data". So, it is better to merge both of them too. Thus, the "status.order" channel is merged

with the "display.data" channel in version-2.

Fig.6.1. shows the complete PKBZ version-2 modules distribution and the corresponding hardware channels.

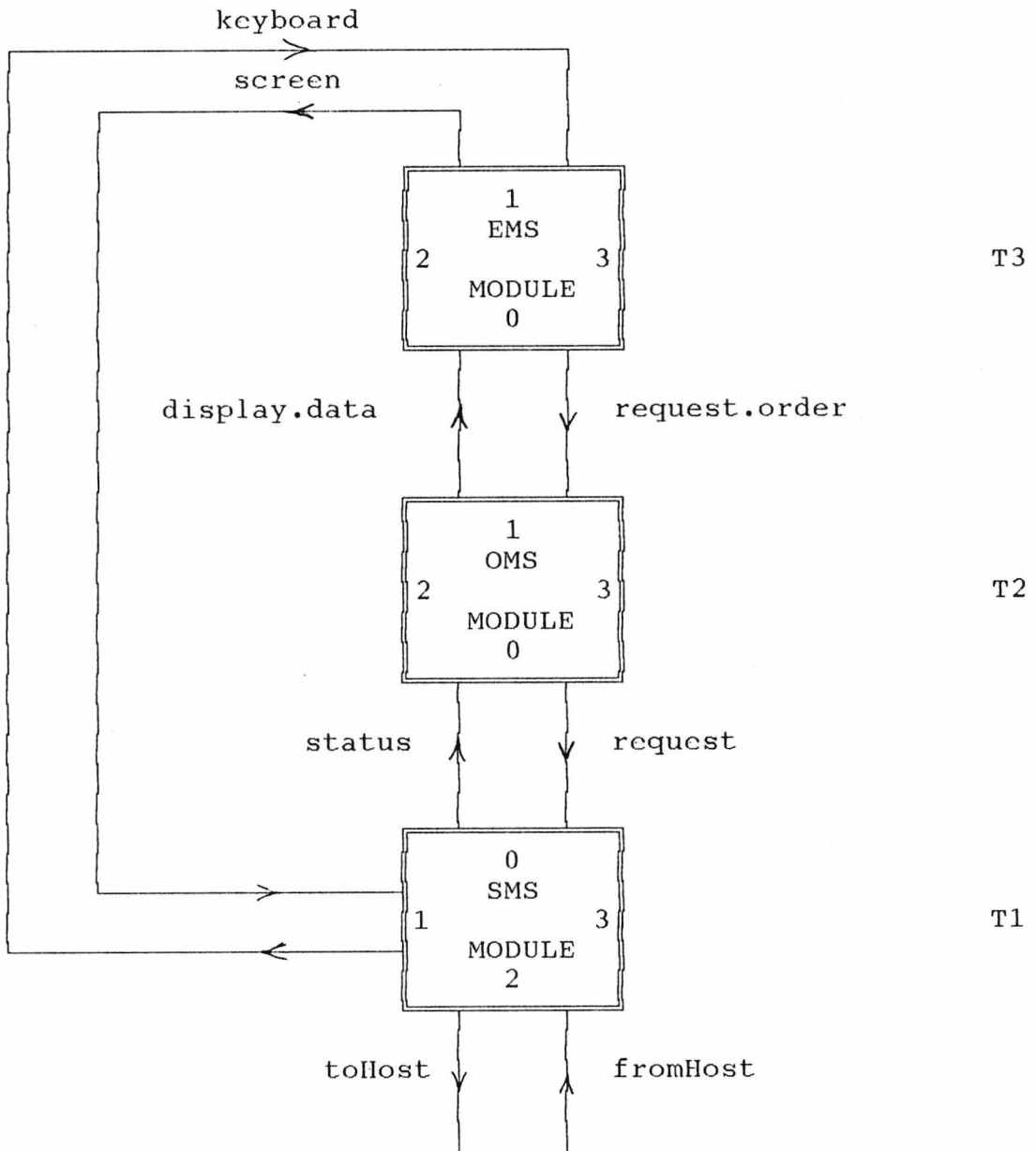


Fig.6.1. PKBZ Version-2 OODBMS Modules Distribution

For the same reason, both the "toOMS" and the "status" channel are merged together. Then, the "fromOMS" channel and the "request" channel form a single hardware channel.

Further, all the system channels are modified to become channel protocols except the channels which are dealing with the peripherals to allow the Occam compiler to check the usage of channels.

The main advantages of the previous modifications are :

1. The unnecessary communication channels are merged with other channels.
2. The channel protocols are checked during the compilation phase to prevent the unnecessary deadlock during the system execution.

6.3.2. Memory Object Communication Channels :

As illustrated in the previous subsection, the number of hardware channels is reduced by merging some channels together. Since there is a correspondence between these channels and the software channels of the Memory Object (see Fig.4.1., and Fig.5.1.), it may be useful to reflect this change in these software channels too. So, the number of input and output channels in the Memory Object is reduced by merging some channels together. Fig.6.2. shows the general Memory Object block diagram in PKBZ version-2.

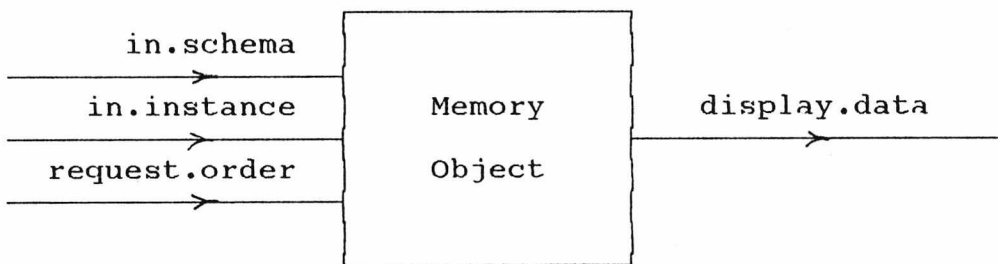


Fig.6.2. PKBZ version-2 Memory Object Block Diagram

Further, all the Memory Object channels are modified to become channel protocols to allow Occam compiler to check the usage of the channels.

The main disadvantage of this merge is that all processes inside the system have to be changed to reflect these changes.

6.4. Memory Object Instances :

In the previous subsections, the major modifications in version-2 have been

discussed. Those changes were made to improve efficiency and do not affect the functionality of the system. In this subsection, the first enhancement will be illustrated; that is, increasing the number of instances allowed in each object class. As mentioned before, in PKBZ version-1, the number of instances per Memory Object is set to the "max.number.of.object". But, in PKBZ version-2, the number of instances can be calculated by the formula :

$$\text{No. of Instances} = \text{INT} \frac{\text{Page Object size} - \text{Page Data Region size}}{\text{Object Instance Size}}$$

This number of instances is stored into the corresponding Memory Object in the OMS module. So, when the number of instances exceeds this number, the instances are transferred into the corresponding Page Object in the SMS module and a new Page Object in the SMS module is allocated together with a new page in the secondary storage database file, and so on. By this mechanism, the number of instances per object class is increased.

6.5. External Management System Batch Processing :

In this subsection, the second enhancement will be described; that is, incorporating batch processing facilities. New EMS modules are designed in version-2 to allow batch processing with the PKBZ system. The main reasons of such design and implementation are :

1. To check the possibility of batch processing
2. To allow the performance evaluation measurements

The new batch modules are designed to implement the following functions :

1. Create different object classes
2. Add instances

A batch process was first implemented using the "ADD.INSTANCE" message (described in chapter 5) which was used to add instances in an object class to allow performance evaluation measurements. Unfortunately, the response time was high during the experiment for the following reasons:

1. During adding each instance, the message checks the existence of the corresponding object classes and gets the location of the corresponding Memory Objects in the OMS module.
2. The updating mechanism using Object Update Parallel processing is carried out periodically with each transaction.

A new batch process was then implemented using new adding messages called "START.ADD", "ADD.INSTANCES", and "ADD.LAST" messages (see Appendix E.). These messages are designed for entity aggregate object. The new batch process, including the measurement of system performance, can be described as follows :

i. In EMS module :

```

PROC Add.Batch.Instance ( ... , ... ,      , ... )
  ... Variables Declaration
  SEQ
    ... Accept entity aggregate object name
    ... Send message START.ADD to the OMS module
    ... Accept status from the OMS module
    ... Accept the number of instances to be added
    ... Start time
    SEQ n = 0   FOR   (number.of.instances - 1)
      SEQ
        ... Generate instance for each structural object
        ... Convert structural instances into a single string
        ... Send ADD.INSTANCES message to the OMS module
        ... Get status from the OMS module
      IF
        (n \ 100) = 0 -- Read Time every 100 instances
          ... Read time
        TRUE
          SKIP
        ... Generate last instance for each structural object
        ... Convert structural instances into a single string
        ... Send ADD.LAST message to the OMS
        ... Get status from the OMS module
        ... Read time
        ... Display all times corresponding to each 100 instances
    :

```

The START.ADD message sends the entity aggregate object name with the related structural object names. Then, ADD.INSTANCES sends object instances in the form of strings in the same order as the names of the structural objects in the START.ADD message. The ADD.LAST is used to indicate that this instance is the last instance in this batch.

ii. In OMS module :

When the START.ADD message is received, the OMS module realizes that a batch addition will be carried out. Then, the following code expresses the process :

```
... Variables Declaration
SEQ
... Check the entity aggregate object existence in the database
... Send the status to the EMS
IF
  not.exists  -- entity aggregate object does not exists
  SKIP
  TRUE
  SEQ
    ... Load the entity aggregate and
    --- the corresponding structural objects in the main memory
    ... Load the corresponding entity set in the main memory
    loop := TRUE
    WHILE loop
      SEQ
        ... Generate surrogate from the entity set
        request.order ? CASE
          ADD.INSTANCES ; s
          SEQ
            PAR
              ... Add instances in parallel
              --- in all the corresponding structural object
              ... Add instance in entity set
              ... Add instance in entity aggregate object
            ... Send status to the EMS module
            ... Check all the related objects are full
            IF
              FULL
              SEQ
                ... Assign new page to the Memory Object
                ... Transfer instances from Memory Object to
                --- SMS module
              TRUE
              SKIP
            ADD.LAST ; s
          SEQ
            -- Same as ADD.INSTANCES
            loop := FALSE
```

By using these messages, the system checks the existence of the corresponding objects only once. Further, the system prevents the updating mechanism until the Memory Object is full. So, the response time becomes better than using "ADD.INSTANCE" message.

The ADD.INSTANCES message uses the parallel background mechanism to reduce the response time. That is because while the EMS module generates the structural instances, the OMS module checks if the corresponding structural objects are full and generates

a new surrogate.

6.6. PKBZ System Performance Evaluation :

As described before, the system enhancements are necessary in order to carry out experiments with large volumes of data to measure the system performance. In order to measure the PKBZ system performance the following procedures are carried out :

1. The PKBZ system is distributed into all possible different transputer configurations. That is, the necessary modifications of the system channels are carried out to allow the same basic modules to be distributed in various configurations. Table 6.2. illustrates the different cases of distribution of the three basic modules across the transputers.

Transputer	Number of Transputers	T1	T2	T3
CASE				
1	3	SMS	OMS	EMS
2	2	SMS	OMS+EMS	
3	2	SMS + EMS	OMS	
4	2	SMS + OMS	EMS	
5	1	SMS+OMS+EMS		

Table 6.2. Distribution of The PKBZ System Modules Across Transputers

2. The addition process is chosen as an experimental test to measure the PKBZ system performance. That is because, in most database application systems, the addition process has the largest response time.

3. The instances are added using the add batch messages (described in subsection 6.5.) to measure the number of instances during addition processing versus the time.

4. The STAFF entity aggregate and the related inherit objects are chosen as a batch test for addition experiment.

5. The instances are generated internally in the EMS module as illustrated in subsection 6.5. It has to be mentioned that in a true batch system, the data will be read

from a file in batch instead of being generated.

6. The experiments are carried out for all the five cases in Table 6.2.

Table 6.3. illustrates the interrelation between the number of instances during addition versus the time in seconds (sec) for the five cases.

CASE	1	2	3	4	5
Instances					
100	0.262	0.267	0.426	1.382	1.374
200	0.973	1.063	1.377	3.552	3.740
300	1.793	1.888	2.408	6.065	6.270
400	3.095	3.211	3.936	9.239	9.475
500	4.472	4.593	5.540	12.511	12.577
600	6.370	6.489	7.623	16.414	16.478
700	8.304	8.428	9.780	20.299	20.357
800	10.737	10.929	12.451	24.868	24.930
900	13.228	13.425	15.181	29.350	29.315
1000	16.265	16.452	18.390	34.682	34.489

Table 6.3. Relation Between Number of Instances Versus Time In Sec.

Fig.6.3. shows the graphical representation of table 6.3. In Fig.6.3, one can conclude the following :

1. Distribution of the system on three transputers (i.e. case-1) is the best distribution.
2. It is predicated that both the case-1 and case-2 are near to each other, since after the EMS module sends the adding message to the OMS module, the EMS module is idle waiting the status. Thus, the processing load of the EMS module is small.
3. Although the same system is distributed on two transputers in case-2, case-3, and case-4, the figure shows that case-2 is the best, while case-4 slows the system performance. The slow performance in case-4 is due to an imbalance problem, since :
 - a. Both the SMS module and the OMS module are obviously processing heavily, but they are allocated to a single transputer.

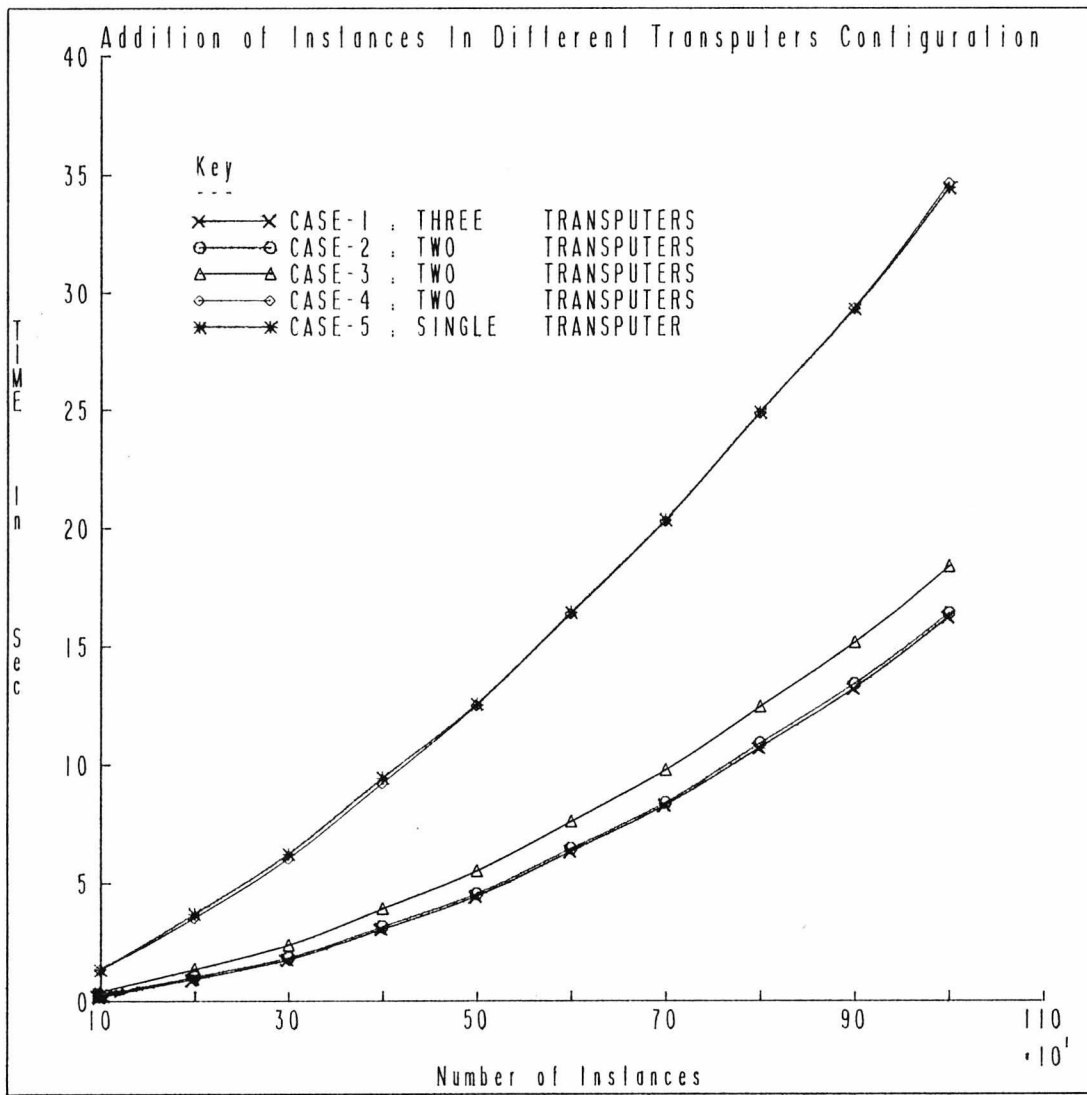


Fig.6.3. Relation Between Number of Instances Versus Time In Second

- b. The EMS module has a small load of processing, but it is allocated into a single transputer.
4. In case-4, although the system is distributed on two transputers, the performance on single transputer (case-5) is better than case-4. This poor performance is due to the bad distribution of modules. Moreover, it indicates that the communication time, in case-4, is greater than the processing time of the EMS module. That is why the system in case-4 is slower than the system in case-5.
5. From case-2 and case-3, one can conclude that the SMS module is processing more heavily than the OMS module. That is because, by adding the EMS module to the OMS module in case-2 the performance is better than adding the EMS module to the SMS module in case-3.
6. Thus, the work load across the different modules is as follows :
- EMS module has the lowest process loading (From case-1 and case-2)
 - SMS module has the highest process loading (From case-2 and case-3)
 - OMS module has a smaller work load than SMS module

This result is expected, since the SMS module is dealing with the mechanical motion of the head movement of the system disk.

It has to be mentioned that the repetition of the same experiment does not give the same results. That is due to the mechanical motion of the hard disk head. So, the previous tables are the average results of five consecutive measurements.

However, there is another parameter which is used to measure the performance of any parallel system. This parameter will be illustrated in the next subsection.

6.6.1. PKBZ Speed Up :

The speed up parameter is defined for each number of processors "n" as the ratio of the elapsed time when executing a program on a single processor (the single processor execution time) to the execution time when "n" processors are available [Eager 89]. That

is, the speed up (S_n) is defined as :

$$S_n = T_1 / T_n$$

T_n ... Execution time on "n" processors

In order to illustrate the speed up in performance from the previous experiment both case-3 and case-4 are excluded, since case-2 is the best distribution using two transputers. Table 6.4. illustrates the Speed Up versus the number of transputers after adding 100, 300, 500, 700, and 1000 instances, while Fig.6.4. shows the graphic representation of the Speed Up.

Number of Instances	100	300	500	700	1000
Number of Trans.					
1	1.000	1.000	1.000	1.000	1.000
2	5.146	3.321	2.738	2.415	2.096
3	5.244	3.497	2.812	2.451	2.120

Table 6.4. Relation Between Speed Up Versus The Number of Transputers

In Fig.6.4., one can conclude that :

1. The speed up is increased sharply by increasing the number of transputers from one to two, while the speed up is increased slightly by increasing the number of transputers from two to three.
2. The decrease in the speed up with respect to the number of instances is due to the hardware communication process.

6.7. Conclusion :

The chapter illustrates the major enhancements in the PKBZ version-2 in which the Memory Object is split into seven different types according to the object classes and the instance types. Moreover, more than one page in the secondary storage database file is allocated to each object class instances.

New EMS modules are designed and implemented to allow batch processes as well

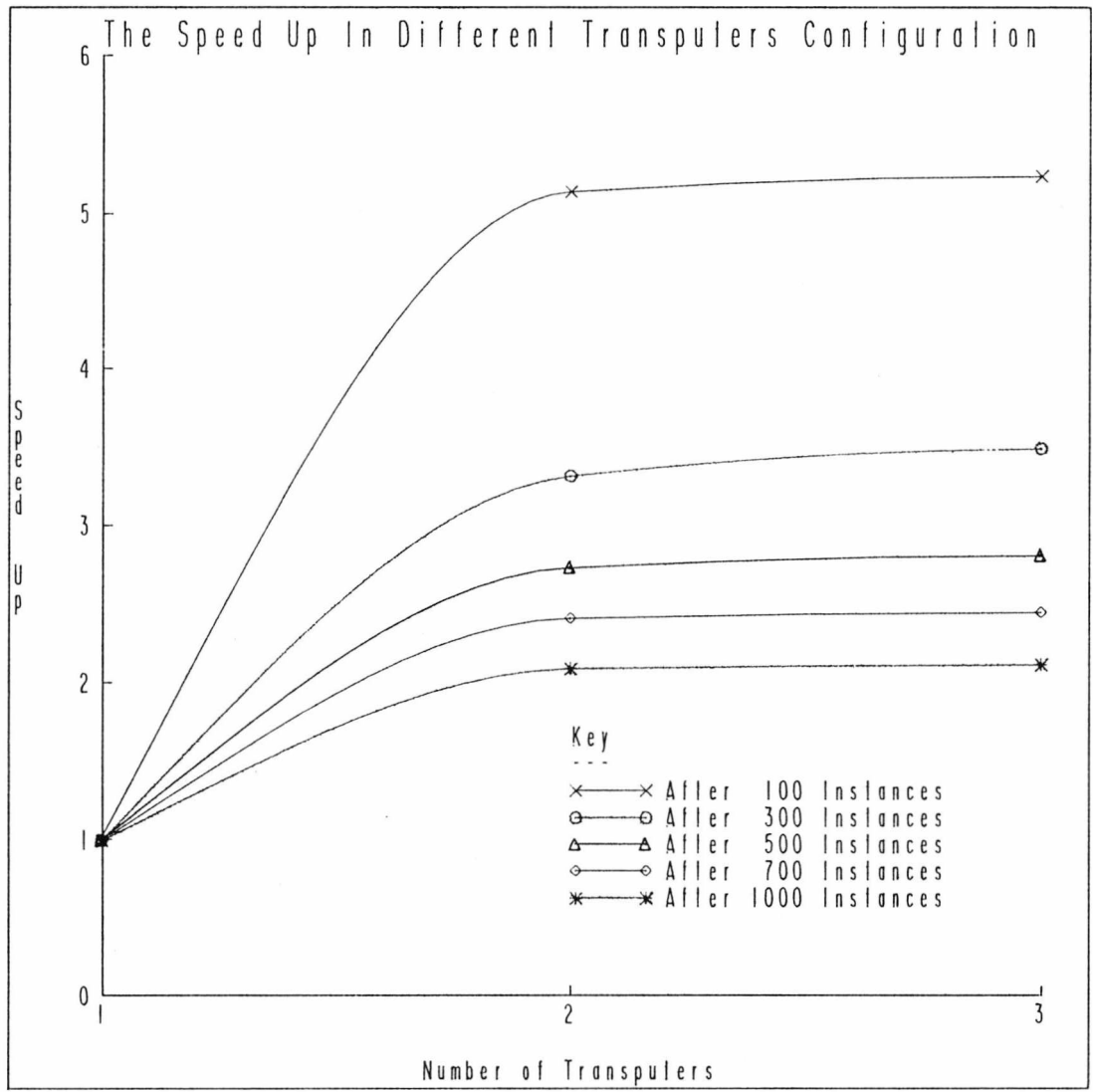


Fig.6.4. Relation Between Speed Up Versus The Number of Transputers

as on-line processes. The experimental results show that there is a significant performance increase in the system using parallel processing, since the Speed Up is increased 2.0 times the speed in a single transputer by distributing the system into two transputers. However, the experiments show that not every distribution increases the system performance. In some cases, the performance becomes poor due to the bad distribution.

CHAPTER 7

Conclusions and Future Researches

7.1. Introduction :

In the previous chapters the design and the implementation of different aspects of parallelism within the scope of object oriented database management systems are illustrated. In this chapter, the general problem areas in both object oriented database management systems and the parallel processing will be described in subsections 7.2., and 7.3. respectively. Then, the conclusions will be illustrated in subsection 7.4., followed by possible future researches in subsection 7.5.

7.2. Problem Areas With an OODBMS :

Generally, the object oriented paradigm presents significant research challenges and an assortment of technical problems. Some of these problems are mentioned in [Oxborrow 89][Zaniolo 86.b], and [Tsichritzis 89]. The problem areas are summarized in the following :

7.2.1. Standard Definitions :

Object oriented systems have inherited their concepts, methods and tools from many other areas in computer science. That is, object oriented systems consist of the repackaging and relabelling of a cross section of different aspects of computer science. As a result, there are disagreements among researchers on basic definitions, for example, "What is an object ?". Researchers coming from programming languages, artificial intelligence, and databases, among other fields have conflicting ideas on when an "object" can be called an object [Hailpern 86]. Therefore, some confusion arises with an object oriented paradigm and this leads to the situation that some of the early OODBMSs are not truly object oriented, but are more like enhancements to the current database systems. However, loose definitions are inevitable during a dynamic period of

scientific discovery. These should become more stable with time.

7.2.2. Performance Problem :

When Codd's [Codd 70] original paper gave a clear specification of a relational database system, there was some criticism that the relational approach would never be commercially viable for performance reasons. But advances in hardware and software technology have resulted in the development of a number of successful relational systems [INGRES 87]. Since an OODBMS must satisfy two criteria : it should be a DBMS, and it should be an object oriented system, the same doubt arises again. However, further advances in hardware and software make it possible that some OODBMSs will become really useful commercial products [Andrews][G-base 88].

7.2.3. Secondary Storage Problems :

Objects are built from simpler objects. The simplest objects are objects such as integers, characters ... etc. Related objects are grouped together in a class in which instances of the class are objects which may be complex in structure. Complexity in object oriented structures presents some challenging storage problems [Zaniolo 86.b].

7.2.4. Schema Evolution :

If the definition or implementation of an object class is changed, this will affect inheriting subclasses, as well as the rules for resolving name clashes in multiple inheritance systems. Supporting modification to class definitions is a difficult problem, especially when the existing instances of the modified classes and subclasses violate the new rules or constraints [Tsichritzis 89]. This problem is well known as "schema evolution" in object oriented systems.

The previous problems are the major problems in OODBMSs. However, [Tsichritzis 89] discusses more sophisticated problems with OODBMSs. Such problems are out of the scope of this research.

7.3. Problem Areas With Parallel Processing :

Although parallel processing increases the computer systems capabilities, there are many problems. These problems can be summarized in the following :

7.3.1. Deadlock Problem :

One of the most common parallel programming errors is that of deadlock [Welch 89]. That is, one (or more) process is waiting on an input or output from another process. Such an error may not be detected easily, so in constructing parallel programs for multiprocessor hardware, there is therefore a need for deadlock-free methodologies and/or powerful debugging tools. Unfortunately, there is no such powerful debugging tool yet. The available "Debugger" on the Meiko Computer Surface is still limited. During the implementation phase of this research, most of the deadlock errors could not be discovered using "Debugger" only.

7.3.2. Non-deterministic Behaviour :

A novel aspect of parallel programming is the non-deterministic behaviour of processes running on parallel hardware. A simple example will make the point. Consider the four processes "A", "B", "C", and "D" indicated in Fig.7.1. The process running on processor "A" performs some calculations on its local data and sends the result to process "D". Similarly, other processes "B" and "C" perform some calculations on their local data and send their results to process "D" too. The process in "D" is very simple: the input received from "A", "B", and "C" are assigned to variables "a", "b", and "c", respectively. Although the processes are all correct there is no way of knowing in advance whether "a", "b", or "c" will be received first. Furthermore, even a repeat run of the same programs with the same data in each processor is not guaranteed to produce the same result. The possibility of such non-determinacy is an unavoidable feature of parallel hardware and can be crucially important in certain applications. It is therefore necessary to isolate such non-determinacy and, if ordering is important, precautions must be taken.

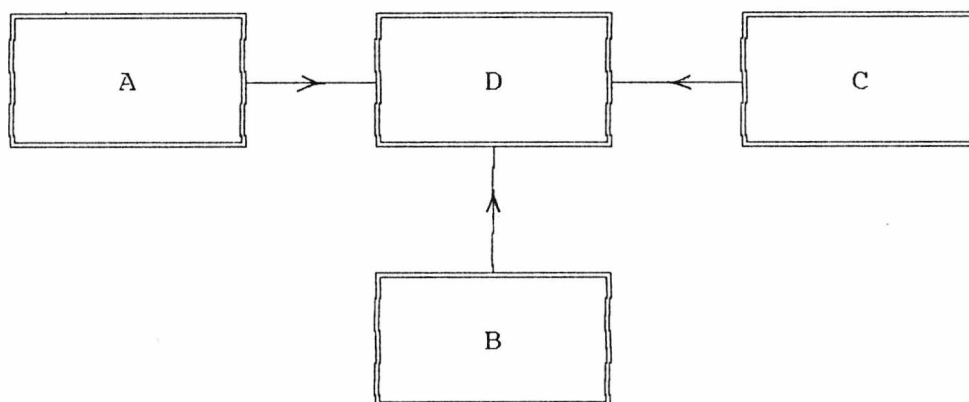


Fig.7.1. Non-deterministic Behaviour

Moreover, the non-determinacy behaviour of parallel processing leads to difficulties in detecting deadlock during the run time, since one cannot know where the deadlock occurs.

7.3.3. Static Language :

Occam is a static language. That is, during the run time, the number of parallel processes cannot be changed. The "PAR" replicator is checked during the compilation phase and the compiler refuses the variable declarations. Such static behaviour leads to difficulties in designing most real life system applications, since the system designer cannot know in advance how many parallel processes are running during the actual run. Such problems can be solved by defining the maximum possible number of allowed parallel processes. However, such a solution results in an overhead during the run time, unless all the parallel processes are executing during the real run.

7.3.4. Pattern Problem :

The transputer's hardwired interconnection must be defined before the run time. However, in some situations, the interconnection depends upon the problem and data which are used during the run time. Such interconnection is fixed and it may be inadequate for some cases with certain data. The actual configuration can be optimized for one particular algorithm by selecting a specific connection pattern. Thus, the same Occam program can be implemented on a variety of transputer configurations, with one

configuration optimized for cost, another for performance, or another for an appropriate balance of cost and performance.

7.3.5. Limited Number of Communication Channels :

Each transputer communicates with the other transputers through hardwired links among transputers. Now, the maximum number of hardwired communication channels is four channel pairs for both input and output. These four channels may be enough for some applications. However, in other applications, with heavy processing load and high communication traffic, four channels are not enough for each transputer to communicate with the external world. This problem can be solved by using "Multiplexer" and "Demultiplexer" processes to increase the number of communication channels by means of software channels. For example, as shown in Fig.7.2., let us suppose that any process of the processes A1, A2, ..., An may communicate with any process of processes B1, B2, ..., Bm and $n, m > 4$.

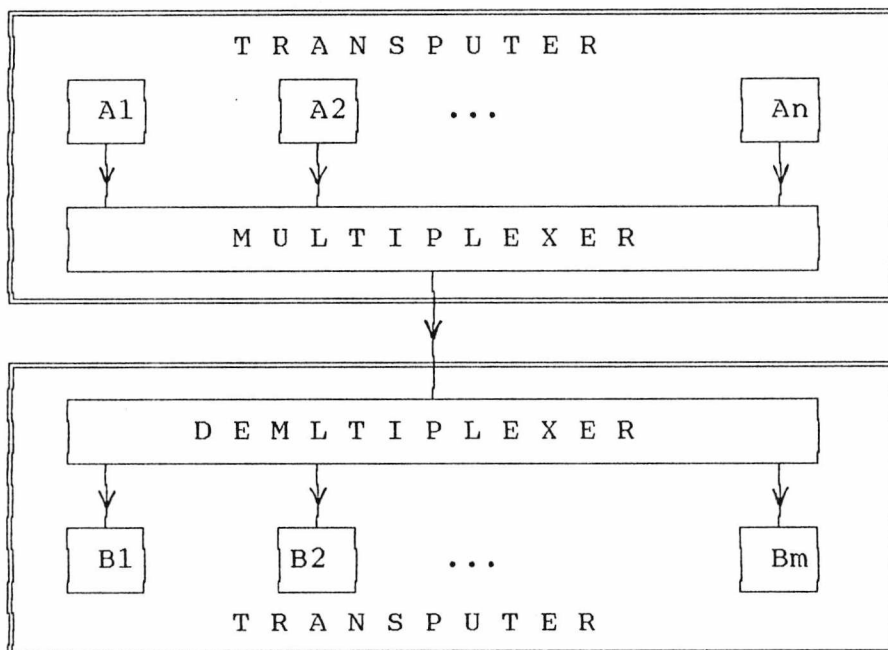


Fig.7.2. Limitation of Communication Channels

Such communication cannot be achieved through hardware channels only. So, the

"Multiplexer" and "Demultiplexer" processes are introduced to allow communication to take place. That is, "Multiplexer" and "Demultiplexer" communicate through a hardware channel while the other processes communicate through software channels inside each transputer. This solution adds extra processes to the system. Moreover, with heavy processing load and high communication traffic, such a solution reduces the processing power of system.

7.3.6. Imbalance Problem :

In a network of running transputers, there can be "load balancing" problems associated with one or more processor being heavily used while another processor or processors are idle for part of the time. This leads to an imbalance problem. Such a problem is not solved easily. That is, the problem of dividing a program among several processors is complex, and typically a situation exists where processors finish their tasks at different times and may be left idle waiting for other processors to complete their tasks. A problem of synchronisation can occur with a processor which is dedicated to controlling the other processors and detecting when the other processors have finished their tasks. A heuristic solution and well-designed system distribution may solve the imbalance problem partially.

7.3.7. Investment Limitation :

Companies already have many hundreds of man-years invested in software developed for sequential, uniprocessor computer systems. Present parallel computers and their compilers cannot run such sequential software without significant modifications. These modifications will require significant investment; hence, it will be some time before they are carried out.

7.3.8. Natural Human Limitation :

As stated before, "as parallel computer systems become more popular, it is important that the system designers and programmers cast off the shackles of sequential thought" [Hey 90]. The human resists a new shape by nature. However, this resistance is

likely to be overcome in future, since parallel processing reflects the semantics of most real life application systems.

7.4. Conclusion :

The thesis has investigated the different aspects and methodologies of parallelism which can be implemented within the scope of the object oriented database management systems. Both the potential of the object oriented approach and the parallel processing were illustrated. The state of the art in related researches were surveyed. A selected number of existing systems which cover both object oriented paradigm and/or parallel processing techniques were investigated.

The main types of parallel aspect, the basic advantages and disadvantages of each aspect together with examples have been illustrated. Descriptions of the different parallel KBZ architectures have been presented.

The development of an experimental KBZ OODBMS prototype on the transputer Meiko Computing Surface has been implemented to test out some of the parallel processing aspects. Two prototype versions have been designed. The current prototypes have been designed and tested in a single user environment. In the current prototypes, the Simple Homogenous Parallel System Architecture has been chosen for implementation. Other aspects of parallelism have been tested too, including Parallel Background. The File Simulation processing has been implemented to check the feasibility of using parallel background technique and to allow fast accessing and updating of data. The principal benefit comes from the ability to perform multiple concurrent simulated disc input/output, thus increasing the disc throughput, and hence improving overall system performance. Further, the File Simulation processing has been implemented in the form of Page Objects in which each transaction is performed via message sending. The current prototypes assume that all Memory Objects are loaded in the main memory using Demanded Object criteria. The other types of Memory Object allocation, deallocation, and grouping have been investigated.

In the current PKBZ prototypes, the Memory Object has been designed in the form

of Occam process so that each Memory Object may be viewed as a complete entity in itself. Both object class schema and the object instances data inside each Memory Object are integrated. Such integration is not supported by the currently available non object oriented databases. Furthermore, both the data and the related methods are encapsulated together. Such encapsulation mechanism provides a way that the system can accumulate the semantic aspects along with data. The Memory Object communicates with the external world through message sending. Each available message is described in detail. The "Parallel Processing Transparency" is satisfied, since the end user does not know how the message is executed in parallel.

By using message sending, both the internal data structures and the data inside the Memory Object are totally isolated. Hence, Memory Object modification and enhancement internally is relatively safe. In addition, the internal representation of data is completely hidden, since only the messages are utilised to access or modify any Memory Object.

The system data has been designed and implemented in the form of objects as with user's data to reflect any enhancement that may occur in the system and to minimize the design and implementation.

In PKBZ version-1, only one Memory Object representation is used for all different types of object class, and a single method name may denote different method implementations according to the object class type. The PKBZ version-1 supports the "overloading" mechanism by choosing the appropriate implementation according to the object type automatically during the run time. The main advantage of this property is that the end user is not aware of the different codes necessary to implement the method.

One of the main modifications in PKBZ version-2 is the Memory Object which is split into different types according to the object classes and the instance types. The prototype has been modified to allow more than one page in the secondary storage database file to be allocated to each set of object class instances. Also, both the software and hardware channels are reduced to minimize the traffic control among processes.

A well defined data structure has been designed for all different object classes. For example, although the structural object class has different property names and types structure than other implemented object classes, a single data structure has been adapted to all different object classes.

A standard data structure has been designed for all entity aggregate instances irrespective of their related structural object classes. Such a standard structure cannot be achieved in any non object oriented database management systems, including relational system. The surrogate values are stored alone so that the corresponding instances data can be instantiated from the related structural object instances in parallel.

The data structure of the Memory Object in the secondary storage is optimal, since the data that have identifiers are suppressed before it is stored into the corresponding secondary storage and it can be fetched when it is needed. The lack of record definition in the Occam language has been overcome by designing a conversion mechanism between the logical data representation and the physical data representation, and vice versa.

The data structure of the database file has been designed to be both homogeneous and optimal. The homogenous nature is reflected in the Page Object which is used as buffer for fast accessing using Object Update Parallel Processing and File Simulation Processing. The database file serves three different groups. Although each group has its own data structure, it is divided into pages. The page is the unit of accessing into the secondary storage. The system stores the detailed structure of each page to prevent the unnecessary mechanical motion of the head. Each page is divided into records and no gap exists at each record in the database file, except at the boundary of the last record in each page.

In version-2, the system modules have been designed and implemented to allow batch processes as well as on-line processes. Results have been obtained which show significant performance improvements when the system is run on small (two to three processors) transputer networks instead of a single transputer, since the response time

is approximately half the time when using three transputers. Further, the Speed Up is 2.0 times the speed in a single transputer by distributing the system into two transputers. However, the experiments show that not every distribution increases the system performance. In some cases, the performance becomes poor due to the bad distribution.

7.5. Future Researches :

There are some further developments in order to improve the system prototype.

The developments can be classified as follows :

- A. Object Oriented Database Developments
- B. Parallel Processing Developments

A. Object Oriented Database Developments :

The system may be modified to include, but not limited to the following :

1. Designing and implementing the other KBZ object classes
2. Introducing index mechanism with data
3. Including functional properties, and constraints
4. Supporting version control
5. Introducing a parallel query processing system
6. Resolving the schema evolution problems
7. Investigating the different allocation and deallocation Memory Object mechanisms
8. Investigating the different types of Memory Object grouping
9. Providing security protection and access control
10. Designing crypt and decrypt mechanisms
11. Introducing data recovery
12. Providing multi-user environment
13. Investigating Multi-database accessing

B. Parallel Processing Developments :

The system may be modified to include, but not limited to the following :

1. Implementing the other types of Parallel on Transputers such as Complex Homogenous Parallel System Architecture and Hybrid Parallel System Architecture. Consequently, Parallel Data Distribution may be implemented.

The development and testing of the prototype PKBZ has confirmed both the functionality and the feasibility of using transputer based parallel hardware for a KBZ OODBMS. Moreover, this work has provided many interesting possibilities for future research.

References

- [Andrews], T. Andrews et al, "The OB2 Object Database" (now known as Ontos; earlier version known as Vbase), Ontologic Inc.
- [Atkinson 89], M. P. Atkinson and et al, "The Object-Oriented Database System manifesto", in Proc. DOODS 1989, Dec. 1989.
- [Bancilhon 88], F. Bancilhon et al, "The design and implementation of O2, an object oriented database system", Proceedings of the OODBS II Workshop, Bad Munster, FRG, Sept. 1988.
- [Banerjee 87], Jay Banerjee et al, "Data Model Issues for Object-Oriented Applications", ACM Transactions on Office Information System, Vol. 5, No. 1, Jan. 1987.
- [Bershad 88], Brian N. Bershad et al, "PRESTO : A System for Object-Oriented Parallel Programming", Tech. Rep. 87-09-01, Department of Computer Science University of Washington, Sept. 1987, Revised Jan. 1988.
- [Birtwistle 73], G. Birtwistle et al, "Simula Begin", Auerbach Press, Philadelphia, 1973.
- [Brookes 89], G. R. Brookes and A. J. Stewart, "Introduction to Occam 2 on the Transputer", ISBN 0-333-45340-9, Macmillan Education Ltd, Houndsmills, London, 1989.
- [Burns 88], A. Burns, "Programming in Occam 2", Addison-Wesley, 1988.
- [Carey 85], M. Carey and D. Dewitt, "Extensible Database Systems", Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems, Feb. 1985.
- [Carey 86.a], M. J. Carey and et al, "The Architecture of the EXODUS Extensible DBMS", Proceeding of International Workshop on Object-Oriented Database Systems, pp 52:65, 1986.
- [Carey 86.b], M. Carey et al, "Object and file management in the EXODUS Extensible Database System", Proceedings of the 12 th VLDB, p. 91:100, Aug. 1986.

- [Carey 87], M. Carey and D. Dewitt, "An Overview of the EXODUS Project", Database Engineering, Vol. 10, no.2, June 1987.
- [Carling 88], A. Carling, "Parallel Processing- Occam and Transputer", Sigma Press, England, 1988.
- [Caruso 87], Caruso and Sciore, "The VISION Object-Oriented Database Management System", Proceedings of the Workshop on Database Programming Languages, Roscoff, France, Sep. 1987.
- [Chalmers 89], Matthew Chalmers, "An Object-Oriented Style For The Computer Surface", Processing of OUG-11 Occam User Group 11th Technical Meeting, 25-26 Sep. 89, Edinburgh, Scotland, Pub. by Ios, Amsterdam, 1989.
- [Codasyl 81.a], Codasyl, Codasyl Data Description Language Committee Journal of Development 1981, Canadian Gov. Publishing Centre, 1981.
- [Codasyl 81.b], Codasyl, Codasyl Cobol Committee Journal of Development 1891, Canadian Gov. Publishing Centre, 1981.
- [Codd 70], E. F. Codd, "A relational model for large shared data banks", Communication of the ACM, Vol. 13, No. 6, p 377:387, June 1970.
- [Cook], Henry Cook, "Applying Parallel Processing to Large Relational Databases", The Teradata DBC/1012.
- [Date 83], C. J. Date, "An Introduction to Database System", Volume II, Addison-Wedley Publishing Company, 1983.
- [Dewhurst 89], S. C. Dewhurst and K. T. Stark, "Programming in C++", Prentice Hall Software Series, Englewood Cliffs, New Jersey 07632, 1989.
- [Dittrich 89], K. Dittrich, "The DAMOKLES Database System for Design Applications: Its Past, its Present, and its Future", in Proceedings 1989, p. 151:171, April 1989.
- [Eager 89], D. L. Eager and E. D. Lazowska, "Speed Up Versus Efficiency in Parallel Systems", IEEE Trans. On Comp., p. 408 : 423, Vol. 38, No. 3, March 1989.

- [Fishman 87], D. H. Fishman et al, "Iris : An Object-Oriented Database Management System", ACM Transactions on Office Information Systems, vol. 5, no.1, pp. 48:69, January 87.
- [Fishman 89], D. H. Fishman et al, "Overview of Iris DBMS" Object-Oriented Concepts, Databases, and Applications, Addison-Wesley, pp. 219:250, (edited by Won Kim 1989)
- [Flynn 86], M. J. Flynn, "Parallel Architectures", System International, Nov. 1986.
- [Ford 88], Steve Ford et al, "ZEITGEIST : Database Support for Object Oriented Programming", Proceedings of the Second International Workshop on Object-Oriented Database System, Sept. 27-30, 1988.
- [G-Bass 88], "G-base version 3, Introductory guide", Graphael, 1988.
- [Gray 85], P. Gray, "Efficient Prolog Access to Codasyl and FDM Databases", in Proceedings of ACM SIGMOD 85, p. 437:443, 1985.
- [Gray 88], P. Gray et al, "A Prolog Interface to a Functional Data Model Database", in Proceedings of International Conference on Extending Database Technology, March 1988.
- [Goldberg 83], A. Goldberg and D. Robson, "Smalltalk-80: The Language and its Implementation", Addison-Wesley, Reading, MA, 1983.
- [Goldberg 84], A. Goldberg, "Smalltalk-80: The Interactive Programming Environment", Addison-Wesley, 1984.
- [Hailpern 86], B. Hailpern and V. Nguyen, "A Model For Object-Based Inheritance". Proceeding Object Oriented Programming Workshop, IBM and Brown University, York Town Heights, SIGPLAN Notices, Oct. 1986.
- [Harland 86], David M. Harland, "REKURSIV - An Architecture for Artificial Intelligence", Proceedings "AI Europe", Wiesbaden, Sept. 1986 with Gunn, Pringle & Beloff.
- [Harland 87], David M. Harland, "OBJEKT - A persistent Object Store", ACM Sigplan, Mid 1987.

- [Harland 88], David M. Harland, "REKURSIV : Object-Oriented Computer Architecture", Ellis Horwood Ltd, 1988.
- [Hayes 87], I. Hayes, "Specification Case Studies", Prentice-Hall, 1987.
- [Hey 90], Tony Hey, "Parallel Processing - A window onto a new world ? ", The Computer Bulletin, Vol. 2, Part 2, 1990.
- [Hockney 81], R. W. Hockney and C. R. Jesshope, "Parallel Computers", Adam Hilger Ltd., U. K., Bristol, 1981.
- [Hornick 87], M. Hornick and S. Zdonik, "A Shared Memory System for an Object-Oriented Database", ACM Transaction on Office Information Systems, vol.5, no.1, p. 70:95, Jan. 1987.
- [IEEE 85], Database Engineering, IEEE Computer Society, vol. 8, no. 4, December 1985 special issue on Object-Oriented Systems (edited by F. Lochovsky)
- [IMS], Information Management System / Virtual Storage General Information Manual, IBM Form No. GH20-1260. IBM Corporation.
- [INGRES 87], INGRES/VMS manuals, Relational Technology Inc. (RTI), 1987.
- [INMOS 84], "OCCAM programming manual"; INMOS Limited England Cliffs; London: Prentice-Hall, International, 1984.
- [INMOS 87], "IMS T800 Architecture", Technical note 6, Inmos Bristol, March 1987.
- [INMOS 88.a], "Transputer Reference Manual", Inmos document number : 72 TRN 006 04, Prentice Hall International (UK) Ltd, 1988.
- [INMOS 88.b], System Software Reference Manual, Meiko, May 1988.
- [INMOS 89], Occam Programming System Manual, Meiko, 1989.
- [Janson 85], "Operating System Structure and Mechanisms", Academic Press Inc. (LONDON) LTD., 1985
- [Jones 87], G. Jones, "Programming in occam", Englewood Cliffs; London : Prentice-Hall, 1987.
- [Jones 88], G. Jones, M. Goldsmith, "Programming in occam 2", Prentice Hall International (UK) Ltd, 1988.

- [Kernighan 88], B. W. Kernighan and D. M. Ritchie, "The C programming Language", Prentice-Hall, Englewood Cliffs, NJ, 1978.
- [Kerridge 87], Jon Kerridge, "A proposal for a Dynamically Reconfigurable Array of Transputers to support database applications", OPPT, 7th Occam User Group and Workshop on parallel programming of Transputer based machine, Sept. 14-16, 1987.
- [Kerschberg 86], L. Kerschberg, Proceedings of First International Workshop on Expert Database Systems, 1986.
- [Kim 87], Won Kim et al, "Operations and Implementation of Complex Objects", Proceedings of the Data Engineering Conference, Los Angeles, CA, 1987.
- [Kim 89], Won Kim et al, "Features of the ORTON Object-Oriented Database System", Object-Oriented Concepts, Databases, and Applications, Addison-Wesley, pp. 251:282, (edited by Won Kim 1989).
- [King 84], R. King, "A Database Management System Based on the Object Oriented Model", Proc. Int. Workshop on Expert Database Systems, p. 443:468, Oct. 1984.
- [Lindsay 87], B. Lindsay et al, "A Data Management Extension Architecture", Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Francisco, CA, 1987.
- [Lorie 83], R. Lorie and W. Plouffe, "Complex Objects and Their Use in Design Transactions", Databases for Engineering Applications, Database Week (ACM), pp. 115:121, May 1983.
- [Oxborrow 88.a], E. A. Oxborrow, "KBZ an object-oriented approach to the specification and management of knowledge bases", in Proceedings of the Sixth British National Conference on Databases (BNCOD-6), 1988.
- [Oxborrow 88.b], E. A. Oxborrow, "Object-Oriented Database Systems - What are they and what is their future", in Database Technology, vol. 2, no. 1,

- p. 31:39, 1988.
- [Oxborrow 89], E. A. Oxborrow, "Databases and Database Systems", Chartwell-Bratt, 2nd edition, 1989.
- [Pountain 88], Dick Pountain, "Rekursiv : An Object-Oriented CPU", BYTE, pp 341:349, Nov. 1988.
- [Purdy 87], A. Purdy et al, "Integrating an Object Server with Other Worlds", ACM Trans. on Office Information Systems, vol. 5, no. 1, Jan. 1987.
- [Rowe 87], L. Rowe and M. Stonebraker, "The POSTGRES Data Model", Proceedings of the 13th International Conference on Very Large Data Bases, Brighton, England, 1987.
- [Schaffert 86], C. Schaffert et al, "An Introduction to Trellis/Owl". ACM SIGPLAN Notices - Proceedings of OOPSLA'86, Vol. 21, No. 11, Nov. 1986.
- [Schwarz 86], P. Schwarz et al, "Extensibility in the Starburst Database System", Proceedings of the International Workshop on Object-Oriented Database Systems, Pacific Grove, CA, September 1986.
- [Stonebraker 86], M. Stonebraker and L. Rowe, "The design of POSTGRES", Proceedings of the 1986 SIGMOD Conference, Washington, DC, May 1986.
- [Stroustrup 86], B. Stroustrup, "The C++ Programming Language", Addison-Wesley, March 1986.
- [Sun 86], The Sun Workstation Architecture, Sun Microsystems Inc., Mountain View, CA, 1986.
- [Thearle 89], R. Thearle, "Survey of Object-Oriented Databases", Zenith Technical Report, University of Kent at Canterbury, 1989.
- [Tsichritzis 89], D. C. Tsichritzis and O. Nierstrasz, "Directions in Object Oriented research", Object-Oriented Concepts Databases, and Applications, Addison-Wesley, p 523:536, (edited by Won Kim 1989)
- [Weiser 89], Steven P. Weiser et al, "OZ+ : An Object-Oriented Databases System", Object-Oriented Concepts Databases, and Applications,

- Addison-Wesley, pp 309:337, (edited by Won Kim 1989)
- [Welch 88], P. H. Welch, "An Occam Approach to Transputer Engineering",
Proceedings of the 3rd Conference on Hypercube Concurrent Computers
and Applications", Pasadena, California, U.S.A., 19-20 Jan., 1988.
- [Welch 89], P. H. Welch, "TRANSNET - A Transputer-Based Communications Service".
In 'Applying Transputer-Based Parallel Machines', Proceedings of the
10th Occam User Group Technical Meeting, Enschede, Netherlands;
pp. 198:212. Published by IOS, Netherlands (ISBN 90 5199 011 1).
April, 1989.
- [Zaniolo 86.a], C. Zaniolo, "Prolog: A Database Query Language for All Seasons",
Proceedings of First International Workshop on Expert Database
Systems, 1986.
- [Zaniolo 86.b], C. Zaniolo et al, "Object Oriented Database Systems and
Knowledge Systems", Proceeding of First International Workshop
on Expert Database System, p 49:65, 1986.
- [Zdonik 85], S. Zdonik and P. Wegner, "A Database Approach to Languages,
Libraries and Environment", Brown University Technical Report No.
CS-85-10, May 1985.
- [Zdonik 86], S. B. Zdonik and P. Wegner, "Language and methodology for object-
oriented database environments. In Proceeding of the Nineteenth
Annual Hawaii International Conference on System Sciences Honolulu,
Jan. 1986, pp 378-387.

APPENDIX (A)

PKBZ Object Oriented Database Management System

Constants and Variables Description

A.1. Miscellaneous Variables Description and Values :

The following table illustrates miscellaneous variables and the corresponding values used in designing and implementing the PKBZ system :

Component Name	Value
object.name.length	12
max.number.of.property property.name.length property.type.length	10 object.name.length object.name.length
max.number.of.inherit max.number.of.inherited.by	10 10
max.number.of.constraint constraint.length	10 40
max.number.of.function function.name.length function.property.length	10 12 40
max.number.of.instance. description	8
max.number.of.objects	50
file.name.length fileSys.maxDataBytes (1) FSMDB NO.REC.IN.PAGE number.of.status	11 512 fileSys.maxDataBytes 5 10

- (1) FileSys.maxDatabytes (FSMDB) is a variable name defined by Occam file system library [INMOS 88.b]. It defines the maximum record length which can be used with the file (i.e. 512 Bytes/record).

A.2. Object Type Identifier Values :

The following table represents the corresponding values of each object type identifier in the PKBZ system :

Component Name	Value
entity.type	1
attribute.type.int	2
attribute.type.string	3
structural.type.int.int	4
structural.type.int.string	5
entity.aggregate.type	6

A.3. Instance Type Identifier Values :

The following table represents the corresponding values of each instance type identifier in the PKBZ system :

Component Name	Value
int	1
string12	2
int.string12	3
int.int	4
stringFSMDB	5
entity.aggregate	6

A.4. Physical Instance Size According to Object Type Instance :

The following table represents the physical instance size of each logical object type instance in the PKBZ system :

Component Name	Size in Bytes
int	2
string12	12
int.string12	14
int.int	4
stringFSMDB	512 (1)
entity.aggregate	2

(1) See FSMDB appendix A.1.

APPENDIX (B)

PKBZ Schema Data Structure

The following subsections illustrates the schema data structure in different object classes. Appendix A.1. defines the size of the variable array.

B.1. The Object Identification Data Structure :

The data structure of object identification is defined as follows :

INT	object.type.id:
INT	object.id:
[object.name.length]BYTE	object.name:

The following table provides a brief description of the components involved in the object identifications :

Component Name	Description
object.type.id	The type of object class and the corresponding instance type
object.id	The unique object identifier of object class
object.name	The object name of object class

B.2. The Object Class Property Names and Types Data Structure :

The data structure of the property names and types is defined as follows :

INT	total.no.of.data.properties:
[max.number.of.property][property.name.length]BYTE	property.name:
[max.number.of.property][property.type.length]BYTE	property.type:

The following table provides a brief description of the components involved in the property names and types :

Component Name	Description
total.no.of.data.properties	The number of defined data properties in the object class schema
property.name	The property name defined in the object class schema
property.type	The corresponding property type of the defined property name

B.3. The Object Class Inheritance Information Data Structure :

The data structure of the inheritance information is defined as follows :

[max.number.of.inherit]INT	inherits:
[max.number.of.inherit][object.name.length]BYTE	inherits.name:
[max.number.of.inherited.by]INT	inherited.by:
[max.number.of.inherited.by][object.name.length]BYTE	inherited.by.name:

The following table provides a brief description of the components involved in the inheritance information :

Component Name	Description
inherits	The inherit object classes identifiers
inherits.name	The inherit object classes names corresponding to object identifiers
inherited.by	The inherited by object classes identifiers
inherited.by.name	The inherited by object classes names corresponding to object identifiers

B.4. The Object Class Constraints Data Structure :

The data structure of the object class constraints information is defined as follows :

```
INT                                     total.no.of.constraints:
[max.number.of.constraint]INT          constraint.no:
[max.number.of.constraint][constraint.length]BYTE  constraint:
```

The following table provides a brief description of the components involved in the constraint information :

Component Name	Description
total.no.of.constraints	The number of defined constraints lines in all defined constraints
constraint.no	The number of lines in each constraint
constraint	The content of each constraint line

B.5. The Object Class Function Properties Data Structure :

The data structure of the function properties information is defined as follows:

```

INT
[max.number.of.function]INT
[max.number.of.function][function.name.length]BYTE
[max.number.of.function][function.property.length]BYTE
total.no.of.functions:
function.no:
func.name:
functions:
    
```

The following table provides a brief description of the components involved in the function properties information :

Component Name	Description
total.no.of.functions	The number of defined function lines in all defined functions
function.no	The number of lines in each function
function.name	The function name defined by user
functions	The content of each function line

B.6. The Instances Description of Object Class Data Structure :

The data structure of the instances description information is defined as follows:

[max.number.of.instance.description]INT instance.description:

The following table provides a brief description of each element in the instance description :

Component Name	Description
instance.description[0]	Previous page number of object instance on the database file
instance.description[1]	Current page number of object instance on the database file
instance.description[2]	Free location index within the current page number
instance.description[3]	Total number of instances
instance.description[4]	New surrogate value
instance.description[5]	Reserved
instance.description[6]	Reserved
instance.description[7]	Instance type (see appendix A.3)

C.4. The Data Structure of The Entity Aggregate Object Instances:

The data structure of the instances information is defined as follows :

[max.number.of.objects]INT id.instance:

The following table provides a brief description of the instance :

Component Name	Description
id.instance	Common surrogate instance is stored in sequence in each element of array

The value of the surrogate (common domain instance) is used to link all the inherit structural objects. So, the instances of entity aggregate are instantiated during the system life.

C.5. The Data Structure of The Page Object Instances:

The data structure of the instances information is defined as follows :

[NO.REC.IN.PAGE][FSMDB]BYTE record:

The following table provides a brief description of the instance :

Component Name	Description
record	Two dimension string array to store page in the Page Object

C.6. The Data Structure of The Page Data Region in Page Object

Instances :

The data structure of the Page Data Region information is defined as follows:

[8]INT

Page.Data.Region:

The following table provides a brief description of the function of each element in the Page Data Region :

Component Name	Description
Page.Data.Region[0]	Number of stored records per page
Page.Data.Region[1]	Number of instances in this page
Page.Data.Region[2]	Current page number
Page.Data.Region[3]	Previous page number
Page.Data.Region[4] to Page.Data.Region[7]	Reserved

APPENDIX (D)

PKBZ Protocols Descriptions

The following protocols definitions are used in PKBZ Object Oriented Database System :

D.1. Protocol REQUEST.ORDER Definition :

The data structure of the protocol REQUEST.ORDER description is defined as follows :

```
PROTOCOL REQUEST.ORDER
CASE
  -- System Message Protocols
  CREATE.DBASE           ; [file.name.length]BYTE
  OPEN.DBASE            ; [file.name.length]BYTE
  END
  -- Create Message Protocols
  CREATE.OBJECT
  CREATE.SCHEMA
    -- Object Identification
    ; INT ; INT
    ; [object.name.length]BYTE
    -- Property Names and Types
    ; INT
    ; [max.number.of.property][property.name.length]BYTE
    ; [max.number.of.property][property.type.length]BYTE
    -- Inherits Data
    ; [max.number.of.inherit]INT
    ; [max.number.of.inherit][object.name.length]BYTE
    -- Inherited By Data
    ; [max.number.of.inherited.by]INT
    ; [max.number.of.inherited.by][object.name.length]BYTE
    -- Constraints Data
    ; INT
    ; [max.number.of.constraint]INT
    ; [max.number.of.constraint][constraint.length]BYTE
    -- Functional Properties Data
    ; INT
    ; [max.number.of.function]INT
    ; [max.number.of.function][function.name.length]BYTE
    ; [max.number.of.function][function.property.length]BYTE
    -- Instance Description Data
    ; [max.number.of.instance.description]INT
  --
  CREATE.INTERMEDIATE.INT      ; INT:[INT]
  CREATE.INTERMEDIATE.STRING  ; INT
    ; [max.number.of.objects][object.name.length]BYTE
  CREATE.INTERMEDIATE.INT.STRING ; INT
    ; [max.number.of.objects]INT
```

```

        ; [max.number.of.objects][object.name.length]BYTE
CREATE.INTERMEDIATE.INT.INT      ; INT
                                ; [max.number.of.objects]INT
                                ; [max.number.of.objects]INT
-- Internal Message Protocols
OPEN.OBJECT                      ; [object.name.length]BYTE
GENERATE.SURROGATE              ; [object.name.length]BYTE
                                ; [number.of.status]INT
NEW.PAGE.NO                     ; INT
UPDATE.INSTANCE                 ; [object.name.length]BYTE
UPDATE.INHERITED.BY.OBJECT      ; INT
                                ; [object.name.length]BYTE
-- Memory Object Message Protocols
OBJECT.NAME.SCHEMA.DESCRPTION ; [object.name.length]BYTE
-- Transaction Message Protocols
ADD.INSTANCE                    ; [object.name.length]BYTE
                                ; [number.of.status]INT
-- Retrieval Instances Protocols
GET.INSTANCE                    ; [object.name.length]BYTE
                                ; [number.of.status]INT
OBJECT.NAME.INSTANCE            ; [object.name.length]BYTE
                                ; [number.of.status]INT
GET.PRI.RANGES                 ; [object.name.length]BYTE
                                ; [number.of.status]INT
GET.INV.RANGES                 ; [object.name.length]BYTE
                                ; [number.of.status]INT
GET.SPE.VALUE                  ; [object.name.length]BYTE
                                ; [number.of.status]INT
                                ; [max.number.of.inherit][object.name.length]BYTE
                                ; [max.number.of.inherit][object.name.length]BYTE
UPDAT.INSTANCE.VALUE           ; [object.name.length]BYTE
                                ; [number.of.status]INT
DELETE.INSTANCE.VALUE          ; [object.name.length]BYTE
                                ; [number.of.status]INT
-- Instance Value Protocols
ID.INSTANCE                    ; INT
NAME.INSTANCE                  ; [object.name.length]BYTE
ID.NAME.INSTANCE               ; INT
                                ; [object.name.length]BYTE
ID.INT                         ; INT ; INT
-- Page Object Message Protocols
GET.PART.OF.INSTANCE           ; INT ; INT
ADD.PART.OF.INSTANCE           ; INT ; INT ; INT::[ ]BYTE
-- Add Batch Protocols
START.ADD                      ; [object.name.length]BYTE
                                ; INT
                                ; [max.number.of.inherit][object.name.length]BYTE
ADD.INSTANCES                  ; [max.number.of.inherit][object.name.length]BYTE
ADD.LAST                       ; [max.number.of.inherit][object.name.length]BYTE

```

The function of each protocol will be described in appendix E.2.

D.2. Protocol INSTANCE Definition :

The data structure of the protocol INSTANCE description is defined as follows :

```
PROTOCOL INSTANCE
CASE
  -- Integer Instances
  int.instance          ;INT:[ ]INT
  -- String Instances
  string.instance      ;
    INT;{max.number.of.objects}[object.name.length]BYTE
  -- Integer + String Instances (Structural Object)
  int.string.instance ;
    INT;{max.number.of.objects]INT;
    INT;{max.number.of.objects}[object.name.length]BYTE
  -- Integer + Integer Instances (Structural Object)
  int.int.instance     ;INT;{max.number.of.objects]INT
    {max.number.of.objects]INT
  -- Page Object Instances
  string.FSMDB         ; INT ; [NO.REC.IN.PAGE][FSMDB]BYTE
:
```

The function of each protocol will be described in appendix E.3.

D.3. Protocol OBJECT.SCHEMA Definition :

The data structure of the protocol OBJECT.SCHEMA description is defined as follows :

```
PROTOCOL OBJECT.SCHEMA
CASE
    object.schema                -- tag protocol
    -- Object Identification
    ; INT ; INT
    ; [object.name.length]BYTE
    -- Property Names and Types
    ; INT
    ; [max.number.of.property][property.name.length]BYTE
    ; [max.number.of.property][property.type.length]BYTE
    -- Inherits Data
    ; [max.number.of.inherit]INT
    ; [max.number.of.inherit][object.name.length]BYTE
    -- Inherited By Data
    ; [max.number.of.inherited.by]INT
    ; [max.number.of.inherited.by][object.name.length]BYTE
    -- Constraints Data
    ; INT
    ; [max.number.of.constraint]INT
    ; [max.number.of.constraint][constraint.length]BYTE
    -- Functional Properties Data
    ; INT
    ; [max.number.of.function]INT
    ; [max.number.of.function][function.name.length]BYTE
    ; [max.number.of.function][function.property.length]BYTE
    -- Instance Description Data
    ; [max.number.of.instance.description]INT
:
```

The function of each protocol will be described in appendix E.4.

APPENDIX (E)

PKBZ Message Function and The Data Structure Descriptions

The following subsections illustrate the message functions and the related data structure which are used in the PKBZ system. The common variables data structure will be described in subsection E.1. Then, the data structure of the REQUEST.ORDER messages, INSTANCE messages and OBJECT.SCHEMA messages will be described in subsections E.2., E.3., E.4. respectively. Appendix A.1. defines the size of variable array.

E.1. Common Variables Data Structure :

The data structure of the common variables which is used in the PKBZ messages is defined as follows:

```
-- Schema Variables
INT                object.type.id:
INT                object.id:
[object.name.length]BYTE  object.name:
--
INT                total.no.of.data.properties:
[max.number.of.property][property.name.length]BYTE  property.name:
[max.number.of.property][property.type.length]BYTE  property.type:
--
[max.number.of.inherit]INT                inherits:
[max.number.of.inherit][object.name.length]BYTE  inherits.name:
[max.number.of.inherited.by]INT                inherited.by:
[max.number.of.inherited.by][object.name.length]BYTE  inherited.by.name:
--
INT                total.no.of.constraints:
[max.number.of.constraint]INT                constraint.no:
[max.number.of.constraint][constraint.length]BYTE  constraint:
--
INT                total.no.of.functions:
[max.number.of.function]INT                function.no:
[max.number.of.function][function.name.length]BYTE  func.name:
[max.number.of.function][function.property.length]BYTE  functions:
--
[max.number.of.instance.description]INT  instance.description:
-- Miscellaneous Variables
[file.name.length]BYTE                database.name:
INT                no.of.instances:
[max.number.of.objects][object.name.length]BYTE  string.instances:
[max.number.of.objects]INT                int.instance, int.domain , int.range :
[max.number.of.objects][object.name.length]BYTE  string.range:
[number.of.status]INT                int.status:
INT                page.no:
[NO.REC.IN.PAGE][FSMDB]BYTE                page:
```

```

---
INT                               inherited.by.id:
[object.name.length]BYTE         inherited.by.nam:
[max.number.of.inherit][object.name.length]BYTE inherits.values:
INT                               id.instance , id.domain.instance , id.range.instance:
[object.name.length]BYTE         name.instance:
INT                               start , size , record.no:
[NO.REC.IN.PAGE * FSMDB]BYTE     s:
INT                               no.of.inherits:

```

The "Schema Variables" above have the same names and functions which are described in appendix B. So, the following table will illustrate the function of the "Miscellaneous Variables" only.

Component Name	Description
database.name	Database name
no.of.instances	Number of instances in the message
string.instances	Values of string instances
int.instance	Values of integer instances
int.domain	Values of integer domain instances
int.range	Values of integer range instances
string.range	Values of string range instances
int.status	Integer statuses of object class
page.no	Page Number Value
page	Page Object instances
inherited.by.id	Inherited by object class identifier
inherited.by.nam	Inherited by object class name
inherits.values	Data values of inherit objects
id.instance	Single value of integer instance
id.domain.instance	Single value of domain integer
id.range.instance	Single value of range integer
name.instance	Single value of string instance
start	Index indicating the start position

continue to next page

Component Name	Description
size	Size of the array needed
record.no	Record number in the Page Object
s	Page Object instances in a single array form
no.of.inherits	Number of inherit object classes

E.2. REQUEST.ORDER Messages Data Structure :

The data structure of each message in the REQUEST.ORDER protocol is defined as follows :

```
-- System Message Protocols
CREATE.DBASE                ; database.name
OPEN.DBASE                  ; database.name
END

-- Create Message Protocols
CREATE.OBJECT
CREATE.SCHEMA
    -- Object Identification
    ; object.type.id ; object.id ; object.name
    -- Property Names and Types
    ; total.no.of.data.properties ; property.name
    ; property.type
    -- Inherits Data
    ; inherits                ; inherits.name
    -- Inherited By Data
    ; inherited.by            ; inherited.by.name
    -- Constraints Data
    ; total.no.of.constraints ; constraint.no ; constraint
    -- Functional Properties Data
    ; total.no.of.functions  ; function.no ; func.name ; functions
    -- Instance Description Data
    ; instance.description
CREATE.INTERMEDIATE.INT
    ; no.of.instances:[int.instance FROM 0 FOR
                        no.of.instances]
CREATE.INTERMEDIATE.STRING ; no.of.instances
                        ; string.instances
CREATE.INTERMEDIATE.INT.STRING ; no.of.instances ; int.domain
                        ; string.range
CREATE.INTERMEDIATE.INT.INT ; no.of.instances ; int.domain
                        ; int.range

-- Internal Message Protocols
OPEN.OBJECT                ; object.name
GENERATE.SURROGATE         ; object.name ; int.status
NEW.PAGE.NO                ; page.no
UPDATE.INSTANCE
UPDATE.INHERITED.BY.OBJECT ; object.name ; inherited.by.id
                        ; inherited.by.nam

-- Memory Object Message Protocols
OBJECT.NAME.SCHEMA.DESCRPTION ; object.name
-- Transaction Message Protocols
ADD.INSTANCE                ; object.name ; int.status
-- Retrieval Instances Protocols
GET.INSTANCE                ; object.name ; int.status
OBJECT.NAME.INSTANCE        ; object.name ; int.status
GET.PRI.RANGES              ; object.name ; int.status
GET.INV.RANGES              ; object.name ; int.status
GET.SPE.VALUE               ; object.name ; int.status
                        ; inherits.name ; inherits.values
UPDATE.INSTANCE.VALUE       ; object.name ; int.status
DELETE.INSTANCE.VALUE       ; object.name ; int.status
```

```

-- Instance Value Protocols
ID.INSTANCE           ; id.instance
NAME.INSTANCE         ; name.instance
ID.NAME.INSTANCE     ; id.instance ; name.instance
ID.INT                ; id.domain.instance
                     ; id.range.instance

-- Page Object Message Protocols
GET.PART.OF.INSTANCE ; start           ; size
ADD.PART.OF.INSTANCE ; record.no      ; start
                     ; size::[s FROM 0 FOR size]

-- Add Batch Protocols
START.ADD             ; object.name      ; no.of.inherits
                     ; inherits.name
ADD.INSTANCE         ; inherits.values
ADD.LAST              ; inherits.values

```

The following table provides a brief description of the tag protocol and the function of each message involved in the REQUEST.ORDER protocol:

Component Name	Description
CREATE.DBASE	Create the database
OPEN.DBASE	Open the database
END	Close the database
CREATE.OBJECT	Create Object Class
CREATE.SCHEMA	Create Object Schema
CREATE.INTERMEDIATE. INT	Create new page for entity object or attribute object with integer instance
CREATE.INTERMEDIATE. STRING	Create new page for attribute object (The instance has string instance)
CREATE.INTERMEDIATE. INT.STRING	Create new page for structural object (The range has string instance)
CREATE.INTERMEDIATE. INT.INT	Create new page for structural object (The range has integer instance)
OPEN.OBJECT	Load the object class into the main storage
GENERATE.SURROGATE	Generate new surrogate
NEW.PAGE.NO	Allocated a new page for Memory Object

continue to next page

Component Name	Description
UPDATE.INSTANCE	Start updating Memory Object
UPDATE.INHERITED. BY.OBJECT	Updated inherited by object data
OBJECT.NAME.SCHEMA. DESCRIPTION	Get object schema description
ADD.INSTANCE	Add the following instance in the object class
GET.INSTANCE	Check existence of the following instance in the object class
OBJECT.NAME.INSTANCE	Get all object class instances
GET.PRI.RANGES	Get the ranges of all primary relationships given the domain value
GET.INV.RANGES	Get all the corresponding domains of the primary relationships given the range
GET.SPE.VALUE	Get specific instance values
UPDATE.INSTANCE.VALUE	Update the following instance
DELETE.INSTANCE.VALUE	Delete the following instance
ID.INSTANCE	Integer instance follows
NAME.INSTANCE	String instance follows
ID.NAME.INSTANCE	Integer and string instance follows
ID.INT	Two integer instances follows
GET.PART.OF.INSTANCE	Get part of the record
ADD.PART.OF.INSTANCE	Add part to the record
START.ADD	Start add batch process
ADD.INSTANCES	Add the following instances
ADD.LAST	Add the last instance in batch

E.3. INSTANCE Messages Data Structure :

The data structure of each message in the INSTANCE protocol is defined as follows:

```

int.instance          ; no.of.instances:[int.instance FROM 0 FOR
                                no.of.instances]
string.instance      ; no.of.instances          ; string.instances
int.string.instance ; no.of.instances          ; int.domain
                    ; string.range
int.int.instance     ; no.of.instances          ; int.domain
                    ; int.range
string.FSMDB         ; page.no                  ; page
    
```

The following table provides a brief description of the tag protocol and the function of each message involved in INSTANCE protocol :

Component Name	Description
int.instance	Integer instance for entity object or attribute object
string.instance	String instance for attribute object
int.string.instance	Integer domain and string range instances for structural object
int.int.instance	Integer domain and integer range instances for structural object
string.FSMDB	Page Object instance values

E.4. OBJECT.SCHEMA Messages Data Structure :

The data structure of each message in the OBJECT.SCHEMA protocol is defined as follows :

```
object.schema
-- Object Identification
; object.type.id ; object.id ; object.name
-- Property Names and Types
; total.no.of.data.properties ; property.name
; property.type
-- Inherits Data
; inherits ; inherits.name
-- Inherited By Data
; inherited.by ; inherited.by.name
-- Constraints Data
; total.no.of.constraints ; constraint.no ; constraint
-- Functional Properties Data
; total.no.of.functions ; function.no ; func.name ; functions
-- Instance Description Data
; instance.description
```

The "object.schema" is used as a tag protocol to indicate that the object schema data is following.

APPENDIX (F)

Database System Group Data Structure

The data structure of the Database System Data Group is defined as follows :

```

VAL  INT  no.of.sys.inform          IS      50:
[no.of.sys.inform]INT              sys.info:
    
```

The following table provides a brief description of each element :

Component Name	Description
sys.info[0]	New available Page
sys.info[1]	Number of records in the database system page
sys.info[2]	Index points to the free position in the database system page
sys.info[3]	Number of records in the database schema page
sys.info[4]	Index points to the free position in the database schema page
sys.info[5]	Number of used pages in the system
sys.info[6]	First used page in the system
sys.info[7]	Second used page in the system
sys.info[8]	Third used page in the system
...	...
...	...
...	...

APPENDIX (G)

External Management System Screens In PKBZ

G.1. The Basic Screen :

The following screen (screen-1) illustrates the main screen in the system :

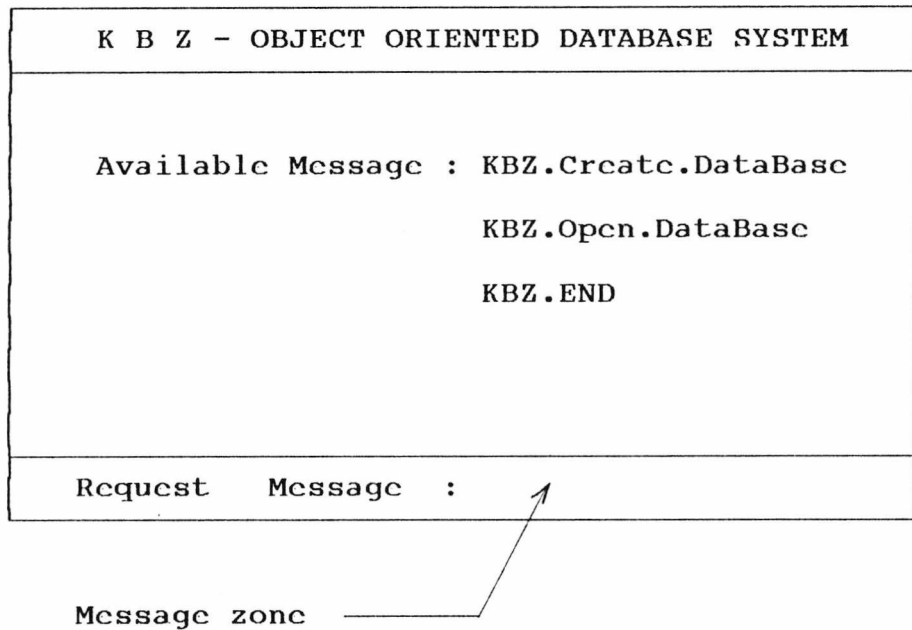


Fig.G.1. Basic PKBZ Screen (screen-1)

G.2. The Message Screen :

The following screen (screen-2) illustrates the message screen in the system :

K B Z - OBJECT ORIENTED DATABASE SYSTEM	
Available Message =====	
ObjectName.SchemaDesc	ObjectName.InstanceVal
ObjectName.AddInstance	ObjectName.GetInsValue
ObjectName.GetPriRange	ObjectName.AddInvRange
ObjectName.GetSpeValue	ObjectName.UpdateIns
ObjectName.DeleteIns	KBZ.CreateObject
KBZ.EndObjectMessage	
Object Name :	Message :

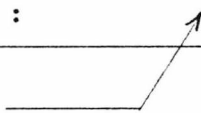
Message zone 

Fig.G.2. Basic PKBZ Message Screen (screen-2)

G.3. The Creation Screen :

The following screen (screen-3) illustrates the creation screen in the system :

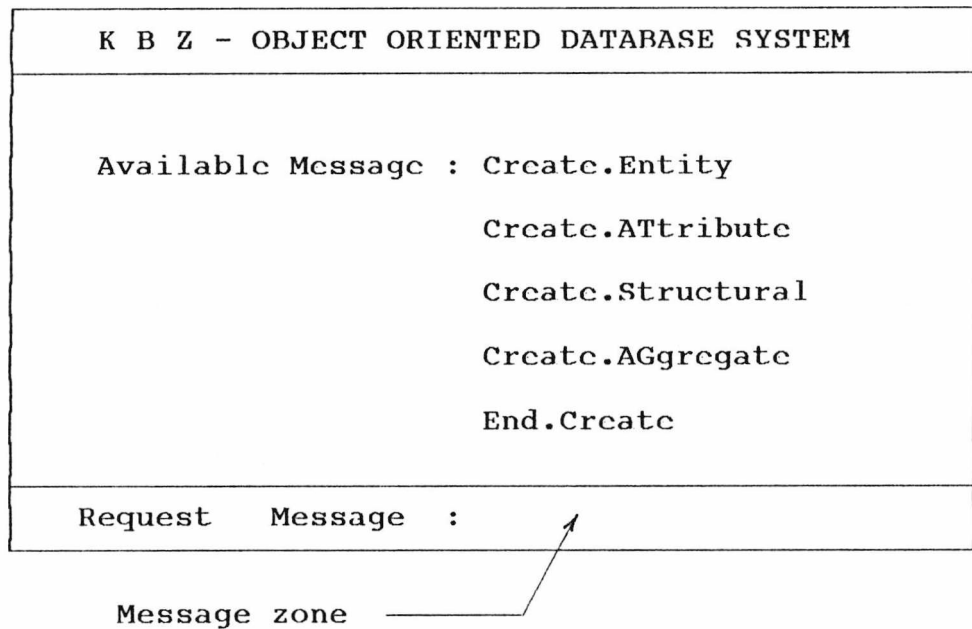


Fig.G.3. The Creation Screen (screen-3)

G.4. Create Structural Object Screen :

The following screen (screen-4) illustrates create structural object screen in the system :

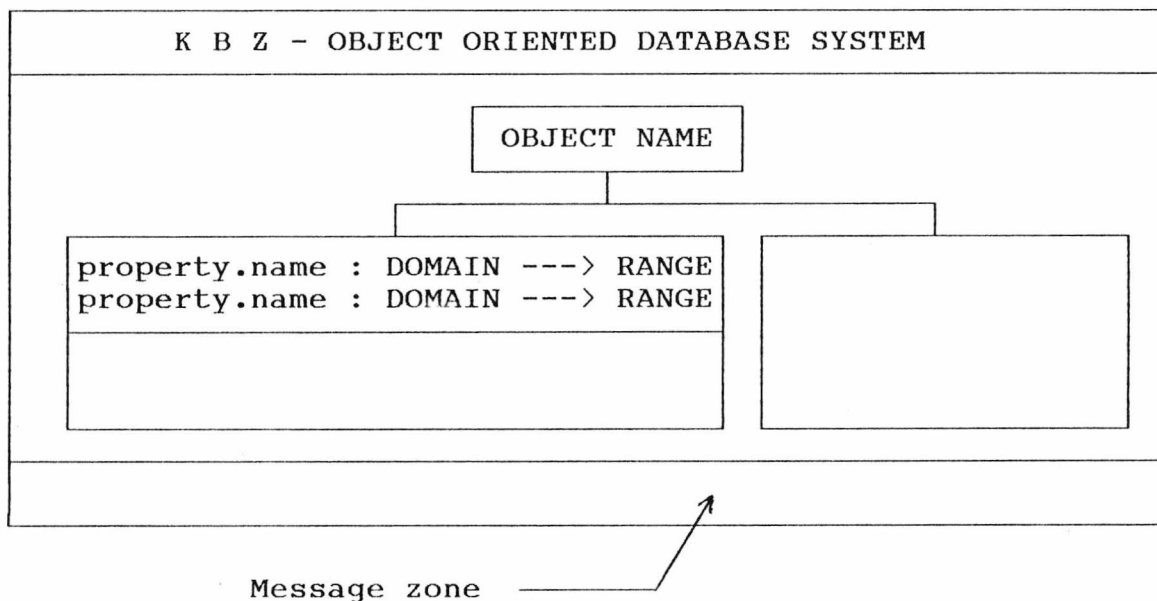


Fig.G.4. The Create Structural Object
Screen (screen-4)

