1991

Exploiting distributed memory architecture to support object-oriented concepts

Kenneth Fakamuria
University of Wollongong

Follow this and additional works at: https://ro.uow.edu.au/theses

Recommended Citation
https://ro.uow.edu.au/theses/2639

Research Online is the open access institutional repository for the University of Wollongong. For further information contact the UOW Library: research-pubs@uow.edu.au
NOTE

This online version of the thesis may have different page formatting and pagination from the paper copy held in the University of Wollongong Library.

UNIVERSITY OF WOLLONGONG

COPYRIGHT WARNING

You may print or download ONE copy of this document for the purpose of your own research or study. The University does not authorise you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site. You are reminded of the following:

Copyright owners are entitled to take legal action against persons who infringe their copyright. A reproduction of material that is protected by copyright may be a copyright infringement. A court may impose penalties and award damages in relation to offences and infringements relating to copyright material. Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.
Exploiting Distributed Memory architectures to support Object-oriented concepts

A thesis submitted in partial fulfilment of the requirements for the award of the degree of

Honours Master of Science
(Computing Science)

from

THE UNIVERSITY OF WOLLONGONG

by


Department of Computer Science
1991
ABSTRACT

This thesis describes an approach to harness the power of the transputer and Occam to support Object-oriented (OO) concepts. It is a specific solution to the more general problem of exploiting distributed memory architectures to support the Object-oriented paradigm. The approach is a mixed one in which a design method based on emerging Object-oriented design methodologies is developed and used to capture OO as well as parallel features of a typical problem domain. The design is then implemented in a non-object-oriented parallel language, Occam, on a network of transputers.

Some testing and evaluation is done on the translation from design to implementation and an assessment is made of the extend to which the approach can be generalised.
# Table of Contents

1. **Introduction**........................................................................... 1

2. **The Object-oriented paradigm and distributed memory architectures**........................................................................... 4
   2.1 Object-oriented concepts................................................... 4
   2.2 Applications of Object-oriented concepts.......................... 9
      2.2.1 Object-oriented programming languages...................... 9
         2.2.1.1 Smalltalk........................................... 9
         2.2.1.2 Common Lisp Object System.................... 11
         2.2.1.3 C++.................................................. 12
         2.2.1.4 Eiffel................................................. 13
      2.2.2 Object-oriented design methods............................... 15
         2.2.2.1 Object-oriented development...................... 15
         2.2.2.2 Object-oriented Requirements Specifications......... 17
         2.2.2.3 Object-oriented systems life-cycle............. 17
         2.2.2.4 A responsibility driven approach........... 19
         2.2.2.5 Object-oriented analysis.................... 21
      2.2.2.6 Summary of object-oriented design methods............. 23
      2.4 Distributed memory architectures..................................... 23
      2.4.1 Characteristics of parallel processing.......................... 24
      2.4.2 Architectures for parallel processing.......................... 26
         2.4.2.1 Issues in Distributed memory architectures........ 27
         2.4.2.2 Distributed memory architecture hardware and software.. 28
      2.4.3 Summary of distributed memory architectures.............. 29

3. **Object-oriented Parallel Systems**........................................ 30
   3.1 Issues in object-oriented parallel systems........................ 30
   3.2 Some parallel object-oriented languages.......................... 32
3.2.1 The ACTOR model ....................................................... 33
3.2.2 POOL-T ........................................................................ 35
  3.2.2.1 Object-oriented ideas in POOL-T .................... 35
  3.2.2.2 Parallelism in POOL-T ................................... 37
3.2.3 POOL and DOOM ........................................................ 37
  3.2.3.1 Object-oriented ideas in POOL ....................... 37
  3.2.3.2 Parallelism in POOL and the DOOM architecture............ 38
3.2.4 Transputers, Objects and Occam ................................ 39
  3.2.4.1 The parallel object model............................. 39
  3.2.4.2 Fusing process and object ......................... 41
  3.2.4.3 Dynamic objects on transputer ...................... 42
3.2.5 Transputers and parallel databases ......................... 43
3.3 Summary of object-oriented parallel systems ......................... 43

4. An approach to exploiting parallel architectures to support object-oriented ideas .................. 45
4.1 And analysis of Object-oriented applications .................... 45
  4.1.1 OOPLs .............................................................. 45
  4.1.2 OOD methodologies ........................................... 46
  4.1.3 Distributed memory architectures ......................... 47
  4.1.4 Object-oriented parallel systems ......................... 47
4.2 The research problem as seen now ................................. 48
4.3 An outline of the approach .............................................. 50

5. An object-oriented design method for an object-oriented parallel application .................. 52
5.1 Introduction ........................................................................ 52
5.2 The design approach........................................................ 53
5.3 The problem domain.......................................................... 55
5.4 Phase I. Finding objects and identifying services ................ 56
  5.4.1 The initial ERD and IFD ................................... 57
  5.4.2 Decomposition of active entities ......................... 59
  5.4.3 Expanding the IFD ........................................... 65
5.5 Phase II. Construct an object interaction diagram and enhance it ........................................... 67
5.6 Phase III. Define class services ........................................ 72
5.7 Phase IV. Map design to target implementation domain ......... 74
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.3.2 Other implementation domains</td>
<td>116</td>
</tr>
<tr>
<td>7.3.3 Other applications</td>
<td>117</td>
</tr>
<tr>
<td>7.4 The development environment</td>
<td>117</td>
</tr>
<tr>
<td>8. Conclusions and Future</td>
<td>119</td>
</tr>
<tr>
<td>8.1 The Design</td>
<td>119</td>
</tr>
<tr>
<td>8.2 The implementation domain</td>
<td>120</td>
</tr>
<tr>
<td>8.3 Future developments</td>
<td>121</td>
</tr>
<tr>
<td>References</td>
<td>123</td>
</tr>
<tr>
<td>APPENDIX A</td>
<td>127</td>
</tr>
<tr>
<td>APPENDIX B</td>
<td>132</td>
</tr>
<tr>
<td>APPENDIX C</td>
<td>134</td>
</tr>
<tr>
<td>APPENDIX D</td>
<td>151</td>
</tr>
<tr>
<td>APPENDIX E</td>
<td>153</td>
</tr>
</tbody>
</table>
ERRATA

The following is a list of typographical errors:

- in Abstract, last line, 'extent'
- p2, line 1, 'extent'
- p2, line 4, 'extent'
- p2, line 7, 'detailed'
- p5, line 15, 'developed'
- p8, line 16, 'changed'
- p9, 4 lines from bottom, remove 'virtually'
- p11, 7 lines from bottom, remove 'a'
- p14, line 4, 'encapsulates'
- p18, line 9, 'Sellers'
- p23, 7 lines from bottom, 'are' not 'is'
- p25, last line, 'discussed'
- p31, 9 lines from bottom, 'that' instead of 'the'
- p42, 2 lines from bottom, add 'is' as last word
- p45, line 5, add 'of' (outline of....)
- p71, line 8, 'created'
- p97, line 4, 'care less'
- p113, line 13, 'a', not 'an'
- p116, line, 3 lines from bottom, 'designed'
- p117, line 7, 'distributed'
- p117, line 1, 'These'
- p123, 1 line from bottom, 'and' not 'ans'
- p125, last reference, 'Tutorial on Distributed-Software Engineering....'
ACKNOWLEDGEMENTS

I would like to thank my supervisor Dr. Jonathan Gray without whose invaluable assistance, encouragement and guidance, this project would not have seen the light of day.

My thanks also go to my wife Masei for her patience and understanding and my daughters Asha and Izadora for their tolerance during the course of this project.
CHAPTER

1

Introduction

Recent developments in the application of Object-oriented (OO) ideas and Distributed memory architectures have had a great impact on software engineering. The object-oriented paradigm has brought about a new approach to software development and opened up a whole new area of research into this field. Object-oriented ideas have for instance found application in programming languages, design, operating systems, database systems and parallel systems to name some of the areas.

On the other hand, rapid advances in distributed memory architectures has amongst other things, made possible the potential for scale up, parallel speed-up in execution and better fault tolerance; features which are desirable in applications such as real time systems and other such time critical applications.

Far apart though they might appear, there are actually some features that both object-oriented concepts and parallel architectures have in common. The object-oriented concepts of message passing and encapsulation for instance are inherent in a parallel application. Given the similarities of object-oriented concepts and distributed memory architectures, and the strengths offered by each, a natural question arises, "how can we exploit these to construct better systems?". This research problem is the subject of many major research studies which will be reviewed in chapter 3.
For this project, the general aim was to determine to what extent we could exploit the parallelism inherent in a distributed memory architecture to support desirable object-oriented features. We were interested in what would be some of benchmarks in the exploitation process and the extend to which we could make generalisations about the translation process. Specifically we wanted to investigate how the transputer and Occam - which together provide an exemplary distributed memory architecture environment, could be exploited to support OO features. In the following chapters, we give a detail description of this investigation and then draw some general conclusions.

In chapter 2, a general survey of the OO paradigm and distributed memory architectures is made. An attempt is made to explain what are characteristic features of the OO paradigm and of distributed memory architectures; it was felt important to identify such characteristics features at the outset to clarify the problem. Applications of the OO paradigm were also investigated; in particular we looked at how object-oriented programming languages (OOPLs) implemented various OO features. We also survey some of the emerging object-oriented design methodologies to see the various representations used to capture OO features.

In chapter 3, we focus onto the problem of developing object-oriented parallel systems. A survey is made of some of the research efforts made into constructing Parallel object-oriented languages (POOLs) and using the transputer/Occam as an implementation domain for object-oriented parallel systems.

In chapter 4, a brief analysis of the research problem is given based on the surveys made in chapters 2 and 3. The problem is then stated more precisely and an outline is made of the approach taken of the investigation.

Chapters 5 and 6 describe the design and implementation of an object-oriented parallel application model. In chapter 5, a model is chosen for the investigation. The design method is then introduced and developed using the model. The idea of the design method is
to capture both object-oriented and parallel features of the model and in the process highlight significant design characteristics for such systems.

In chapter 6 we describe the implementation of the design developed in chapter 5. First we present how OO features were represented in Occam and then how the system was implemented in Occam and distributed onto the transputer network.

In chapter 7, an evaluation is made of the translation process from design to implementation and chapter 8 ends the report with some conclusions drawn from the investigation.

References and appendices are given at the end of the report.
The Object-oriented paradigm and Distributed memory architectures.

In this chapter, aspects of both object-oriented concepts and distributed memory architectures that are of significance to the study will be reviewed in order to put the problem into perspective.

2.1 OBJECT-ORIENTED (OO) CONCEPTS.

Object-oriented concepts emerged as a result of research into new methods of computing. It first appeared with the introduction of SIMULA [Dahl66] and was later developed further in the Small-talk languages; Small-talk-72 and Small-talk80 [Golberg83]. However it only really became significantly more popular in the late 1980s and is now having a profound effect in software engineering.

Central to the object-oriented paradigm is the notion of constructing software systems based on data rather than 'the function'[Meyer88]. An object-oriented system is perceived as
a collection of objects with each object encapsulating its own set of data as well as
behaviours. An object cannot normally access another object's internal state. It can only
interact with it via messages. Such a system enforces information hiding and makes the
complexity of the system more manageable.

In contrast, traditional systems are obsessed with the idea of functional decomposition
and the perception that data and functions exist independently. Functions are there to act on
passive data. Such systems do not support much information hiding and are not responsive to
changes.

The OO paradigm developed in different contexts (Object-oriented language (OOPL)
environments) and as such led to different interpretation of its characteristic features.
However, most authors generally agree that the following features characterise the object-
oriented paradigm, [Cox 86], [Nierstrasz 89], [Meyer 86], [Meyer 88], [Snyder 86], [Blair
etal 91]:

i) Objects and encapsulation.

At the conceptual level, an object represents an entity in the system being develop.
This may be a real world entity or just a conceptual entity in the system. In the
physical sense, it is a run time instance of a class with ".. an encapsulation of a set of
operations which can be invoked externally.." [Blair et al 91].

ii) Classes and instantiation.

The concept of a class is that of a template from which instances of the class can be
created. This template encapsulates operations or methods and data of a typical
instance of the class.

An instantiation mechanism allows multiple instances of the class to be created.
iii) **Inheritance.**

It is often the case that in OO systems, many classes share common data and behaviour. Such classes can be organised into a hierarchy. The semantic notion of *generalisation/specialisation* is used to relate classes in the hierarchy. The **IS-A** relationship relates a class to its superclass as illustrated in fig. 2.1.

![Figure 2.1 A single inheritance hierarchy.](image)

The inheritance hierarchy is defined in the context of this **IS-A** relationship. In this case, a superclass is said to be a *generalisation* of a subclass or subtype and a subclass is a *specialization* of a superclass. Such specialisation can mean that a subclass may inherit all the superclass' data and behaviour and add some new behaviour or inherit from the superclass but override some behaviour. An inheritance hierarchy allows members of the hierarchy to share common behaviour. Single inheritance and multiple inheritance are possible. In single inheritance a class has
only one immediate/direct superclass. An inheritance hierarchy that enforces single
inheritance, results in a tree structure as illustrated in fig. 2.1.

With multiple inheritance, a subclass may inherit from two or more immediate
superclasses as shown in Fig. 2.2.

Fig 2.2. Multiple Inheritance.

Multiple inheritance introduces potential conflicts when two or more methods with the
same name exist in the immediate superclasses. Such conflicts are often difficult to
resolve. [Snyder 86] and [Wegner 87] give further discussion about inheritance and
classification in general.

iv) Message Passing and methods

Message passing is the mechanism by which objects interact with one another. In
object-oriented systems, objects are not normally allowed to access each other’s
methods directly. Instead every object has an external interface which constitutes the
types of services or methods that it can provide. A client object can see a server
object only through its external interface and when it requires the services the object,
it sends a message to it indicating the method required. How the method is performed
is the business of the server object. It is this message passing mechanism that
enforces encapsulation and information hiding.
v) Polymorphism and overloading.

Polymorphism means that a method can have more than one interpretation for different classes in the inheritance hierarchy. Polymorphism can be introduced either through subclassing where a method defined in a class is automatically defined in all its subclasses or though overloading which means that the same name can be used for different methods in different parts of the same inheritance hierarchy. Polymorphism enforces inheritance.

vi) Binding.

The use of polymorphism in an object-oriented system, obviates the need to make the mapping of a method name to its correct method code in the hierarchy. This mapping of method name to method code is referred to as binding. There are two alternatives for binding:-

a) Static binding - In this case binding is done at compile time. The advantages are that there is no run-time overhead in accessing the required code at run time and any binding errors are detected at compile-time. The obvious disadvantage is that bindings cannot be change until the next compile time, which is against what object-oriented systems are all about.

b) Dynamic binding - In this case the mapping of name to code is done on the fly each time the method is invoked. The main advantage is that if the inheritance hierarchy changes dynamically, a desirable feature in OO systems, the bindings can change accordingly to reflect the changes in the hierarchy. The disadvantage is that of the costs incurred in searching for the correct code in the hierarchy at every method invocation.
2.2. APPLICATIONS OF OBJECT-ORIENTED CONCEPTS

Object-oriented concepts have found applications in fields such as programming languages, design methodologies, databases, artificial intelligence, operating systems and distributed computing to name a few. In the following section application of OO concepts in object-oriented programming languages (OOPLs) will be discussed briefly. The objective is to see how various OO features have been implemented and to highlight some of the practical solutions and constraints.

2.2.1 OBJECT-ORIENTED PROGRAMMING LANGUAGES (OOPLS)

Four OOPLs will be compared; Smalltalk, Common Lisp Object System (CLOS), C++ and Eiffel. They will be compared on the basis of how they implement the following categories of OO concepts: i) Objects and classes ii) Encapsulation and methods, iii) Message passing iv) Inheritance and v) Polymorphism and binding.

2.2.1.1 Smalltalk

[Golberg 83] and [Golberg 89] give a thorough description of the Smalltalk language. We summarised below some of the important OO features of the language.

i) Object and class

Virtually every item in Smalltalk is an object including scalar data types such as integers and booleans as well as user defined objects. An object is a runtime instance of a class. The way in which classes are defined in Smalltalk is such that a class is itself also an object. As such, it must itself be an instance
of some other class called a meta class. Both classes and meta classes may have methods. Methods of a metaclass are called class methods. Instances of a class may be created and destroyed dynamically.

ii) Encapsulation and methods.

All variables of a class are private but methods may be private or public.

iii) Message passing.

Message formats identify the object, method to be invoked as well optional parameters. For instance the unary message

\[ \text{angle cos} \]

identifies the method \( \text{cos} \) and the parameter \( \text{angle} \). The method \( \text{cos} \) is defined in a built class called System class.

The keyword message

\[ \text{Flight Addflight : 'QT02'} \]

means invoke method Addflight in class Flight with parameter QT02.

iv) Inheritance

Only single inheritance is supported. A subclass is allowed to modify the superclass' behaviour by adding new methods or overriding any of them.

v) Polymorphism and binding.

The overloading approach is used to enforce polymorphism.
2.2.1.2 Common Lisp Object System (CLOS)

A thorough description of the CLOS is given in [Keene 89], [Kessler 88]. Some important OO features of the language are given here.

i) **Class and object.**

A CLOS class encapsulates only variables which are called *slots*. Methods are defined elsewhere. An object is an instance of a class.

ii) **Encapsulation and methods.**

Methods of a class are encapsulated in generic functions - not in the class. Generic functions group together similar methods of different classes and perform the high level operations in a service request. A class that needs to invoke one of its methods, must call the appropriate generic function with appropriate parameters.

iii) **Message passing.**

Message formats are in the form of *lambda lists*. Each method under a generic function has a unique lambda list which includes the name of the class (and its subclasses) to which the method belongs as well as optional parameters. An object that wishes to invoke a one of its method must send the correct format of the method's lambda list to the appropriate generic function. The lambda list is then checked and the method is invoked if there is found to be a match.

iv) **Inheritance, Polymorphism and binding.**

Both single inheritance and multiple inheritance are supported. Slots and methods are inherited even though methods are kept separately from slots. The selection of appropriate methods in the inheritance hierarchy is all done by
the use of precedence lists. Each class has a precedence list which includes the class itself and all its superclasses. The ordering in the list is from most specific to least specific. The use of precedence list also serves to resolve multiple inheritance conflicts.

2.2.1.3 C++

Details of the C++ can be found in [Stroustrup 86], [Wiener and Pinson 88] and [Weston 90]. A summary of some of C++'s OO features are given below.

i) Objects, classes and encapsulation.

C++ classes are static templates constructed from a modification of the struct data type. As such, they cannot be invoked or passed parameters. An example of a C++ class is shown below.

```cpp
Class ClassZ : public ClassX{
    private: var1, var2, var3:
    protected: var4, var5, var6:
    public:
        method1(par1, par2,...parn)
        method2(par1, par2...parn)
}
```

C++ provides three different levels of access privileges for its variables and methods. Public methods may be accessed from any where, Protected methods or variables are private to the public but may be accessed by subclasses. Private methods or variables are strictly private and can only be accessed by instances of the class.
ii) *Message passing*

C++ uses the dot notation in message calls. Suppose an instance of classX is declared using

```
new ClassZ newclass.
```

A message to `newclass` to invoke `method1` would be

```
newclass.method1(par1, par2, .. parn)
```

iii) *Inheritance.*

The current version of C++ supports both single inheritance and multiple inheritance. ClassZ in the above example is defined here as a subclass of ClassX or a derived class.

Multiple inheritance conflicts are resolved by the use of the `::` (*global scope resolution operator*). Suppose for instance that ClassZ is a derived class of both ClassX as well as ClassY as shown here

```
ClassZ:public ClassX, public ClassY {

...........

};
```

If both ClassX and ClassY have a method call `method1` which ClassZ cannot override then the `::` global scope resolution operator is used to which `method1` to invoke.

### 2.2.1.4 Eiffel

A thorough description of Eiffel can be found in [Meyer 88]. Some important OO features of the language are given here.
i) Classes, objects and encapsulation

Eiffel classes as in C++ are static templates which cannot be invoked or passed parameters.

A class encapsulate data as well as methods. An instantiation mechanism is used to create instances or run time objects of a class. Eiffel defines a certain category of classes called deferred classes. These are defined with partial implementation or no implementation at all. Such classes may be specified by subclasses using a method called assertion. This is one way in which Eiffel is able to enforce polymorphism. Automatic garbage collection is provided in Eiffel. This relieves the programmer from the need to constantly deal with storage management.

iii) Inheritance

Eiffel support multiple inheritance. Inheritance conflicts are resolved by the use of a directed acyclic graph. Such a method prohibits conflicts arising from methods with the same name in superclasses. A subclass will inherit all its superclass's public methods however it may modify superclass behaviour by adding new methods or override some existing ones.

iv) Methods, polymorphism and binding.

Overloading is used to enforce polymorphism. Polymorphic reference may be dynamic and is supported by the use of dynamic binding.
Object-oriented design methodologies are a more recent application domain of OO concepts. The emergence of these object-oriented design methodologies is an important direction of development in OO applications and perhaps a natural one. It has been attributed to the fact that current designs cannot fully express object-oriented features that OOPLs now incorporate [Ormsby 91], [Coad and Yourdon 90]. Meyer [Meyer 88] for instance pointed out that current top-down designs have the following flaws: i) they do not take into account evolutionary changes ii) they are based on the idea of a single function, ".. a questionable concept.." and iii) they do not encourage reusability. Booch in [Booch 86], noted that the development of OODs has been influenced by advances in computer architecture - "..capability systems and support for operating systems", methods of modularisation, abstraction and information hiding.

In the following sections, we review some of the ideas that gave rise to OOD methodologies and highlight important issues.

2.2.2.1 Object-oriented development.

Grady Booch [Booch 86] was one of the first who proposed an object-oriented design method. He noted that the traditional method of functional decomposition, ".. concentrates on major actions of the system and is silent on the issue of agents that perform these actions..". His idea of an object-oriented system consisted of objects that encapsulate behaviours as well as data and interacted via messages. To represent such a system in an OOD, he proposed the following steps:
i) Identify objects and attributes

ii) Identify operations of the object and those it requires from other objects.

iii) Establish visibility of each object in relation to other objects

iv) Establish interface of each object

v) Implement each object.

He pointed out that his method focused only on design and implementation and suggested that structured analysis modelling methods such as the JSD or DFD could be used at the analysis stage. He proposed different methods of graphically representing classes and objects because "DFDs and structure charts do not capture the essence of objects ..". Further he noted that in a large OO system, classes are often arranged into clusters which he referred to as subsystems. Such subsystems are not three dimensional decompositions as in a DFD model. Instead they are structurally flat or two dimensional. Such grouping/clustering makes the complexity of a system more manageable. Booch implemented his OOD model in Ada which although is not an object-oriented language (but an object-based language) supports both concurrency and the object-oriented paradigm to a certain degree. It is interesting to note how OO features of an OO model are implemented. Classes are implemented as packages and objects as instances of packages. Operations are implemented as subprograms exported from a package specification and visibility is statically defined. The concept of inheritance is not used in Ada. In fact Booch maintains that inheritance is an important but not necessary feature of OO systems. Instead, the concept of a generic package which is a parameterised package is used.
2.2.2.2 Object-oriented Requirement Specification.

Booch noted above, that his OOD method was partial and that a structured analysis method such as JSD or DFD could be used as a first stage in the design. Bailin [Bailin 89] proposed a requirements specification method in which both structured analysis and the semantic data model are used to derive an object-oriented model. His approach assumes that a requirements specification textual document exists. This is then used to identify a set of classes or entities, using a method such as proposed by Booch, which would form an initial Entity-Relationship type model. From this model, active entities - those that are of interest at this stage of development, are converted to Extended-DFDs (EDFDs) which are similar to DFDs, and then decomposed to find further entities which may be under each of the current entities. If a new entity is found under an active entity during the decomposition process, it is brought up to the top and added to the current EDFD model thus extending it. New functionality is then added to the new entity. This process of decomposition and search for new entities is an iterative one and would continue until the model is deemed perfect.

Some important points to note in this approach are: i) it is an attempt to derive an OOD using the strengths of structured analysis to identify entity functionality or methods and implicitly the interaction between entities, and the semantic ER model to represent the semantic relationships between entities. ii) In the process of decomposition and finding new entities, clusters of related entities are discovered. Such clusters are important in managing the complexity of the system.

2.2.2.3 Object-oriented system life-cycle

Hendersen-Sellers and Edwards [Hendersen-Sellers 90] noted the shift from the functional decomposition methods which use DFDs to the Jackson Structured Development (JSD)
method [Jackson 83] and then to OOD methodology. They discuss the strengths and
weaknesses of each method and point out that there is a genuine case (i.e heavy investment)
to have a mixed approach to design in which it is possible to have an OOD implemented in a
non-object oriented language or a functional type design implemented in an object-oriented
language or other possible combinations. The partial methods proposed by Booch and Bailin
(sections 2.2.3.1 and 2.2.3.2) they point out are in fact mixed approaches. In any case
current investments on applications that are based on traditional design methodologies is
reason enough to pursue this type of mixed approach. In addition to the different possibilities
of mixed approaches Henderson-sellers notes that current OOD approaches such as OOA
[Coad and Yourdon], class libraries development [Meyer 88], and the Responsibility driven
approach [Wirfs-Brock 90] are increasingly blurring the distinction between phases of the
software life-cycle. From this background and the work done by Booch and Bailin (see
section 2.2.3.1 and 2.2.3.2) he proposes his Fountain model for the object-oriented
software life cycle. This model in essence reflects the overlapping nature of object-oriented
design. He identifies seven steps in the model which are summarised below.

1. Object-oriented system requirements specification using an OOA as in [Coad and
Yourdon 90]
2. Identify objects and their services using methods as in [Meyer 88]
3. Establish interaction between objects using methods as in [Bailin 89] and EDFDs,
ERDs and Information flow diagrams (IFDs) to relate objects.
4. Merge analysis with design phase - use of DFDs/IFDs to identify reusable components.
5. Bottom-up concerns - use of library classes
6. Introduce inheritance relationships as more and more details of the design are
revealed.
7. Introduce aggregation and/or generalization as required.
2.2.2.4 A Responsibility Driven approach.

Wirfs-Brock and Wilkerson, [Wirfs-Brock 89], propose a different approach to OOD which they have called the Responsibility-driven approach. They maintain that current OOD methods that begin by defining objects based on abstract data types have certain weaknesses:

i) They violate the idea of encapsulation by making data structure part of the definition of an object. This makes an object, dependent on structure from the beginning of the design.

ii) Later on, changes to the object's structure will affect other objects that rely on this object's structure.

The responsibility-driven approach they claim improves encapsulation. It uses the idea of clients and servers. Objects are either clients and/or servers. By viewing objects in this manner, objects can be defined in terms of the services or contracts they provide to other objects rather than on abstract data types. The advantage is that it focuses on what the server does and not how it does it. The strengths of this approach are:

i) it leaves the structural details of an object until implementation. ii) Being able to find an object's services at the beginning facilitates polymorphism because these can identify the type of messages or protocols to be used by the object. iii) Objects defined in this manner make it easier to identify object services and hence easier to construct inheritance hierarchies.

In [Wirfs-Brock 90], Wirfs-Brock et al explain the details of the responsibility-driven approach. Since part of the approach has been adopted in this project, we will summarise the basic steps here.

**Initial explorations.**

i) Find classes - class names can be extracted from a requirements specification.

Identify concrete classes

1 9
Identify abstract classes

ii) Determine class responsibilities.

What services does this class provide.

iii) Determine class collaborations.

What other classes does this class interact with.

Detailed Analysis

iv) Determine classification and inheritance

Factor out common classes and create inheritance hierarchies as required.

Subsystems of classes

vi) Identify Subsystems of classes

Cluster classes or groups of classes that work together to provide a subsystem - a clearly defined unit of functionality.

Protocols

vii) Define class protocols.

After these stages, the design is then ready for implementation. As can be seen, the method is very similar to Hendersen-Seller's fountain model. Emphasis however is stressed on identifying a class' responsibility as early as possible. We note also the relative importance they have placed on identifying subsystems or clusters of classes such as those suggested in Booch's model [Booch 86]. Such natural groupings assist in managing the complexity of an OO system.
2.2.2.5 Object-oriented analysis (OOA).

We have already noted [Hendersen-Sellers 90] how in the object oriented life cycle, the distinction between various phases have become increasing blurred. Coad and Yourdon [Coad and Yourdon 90] have proposed an object-oriented analysis model which subsumes parts of the OOD models of Hendersen-Sellers, Wirfs-Brock and Booch. This is further evidence of the blurring of phases in the object-oriented life cycle. We will highlight some of the features of this OOA model which have added to a what is appearing to be a set of common characteristics of an object-oriented design. Coad and Yourdon's OOA model is derived from the semantic data model and OOPLs; concepts such as generalisations/specialisation, whole-part and instance connections were borrowed from the semantic data model while encapsulation, messages and inheritance were borrowed from OOPLs. The model was developed out of a motivation to construct design models that more reflect problem domains, that are consistent, and easily modifiable. It consists of five basic layers; the subject layer, the class & object layer, the structure layer, the attribute layer and the services layer. Each of the five layers will be explained below in the order in which the design proceeds.

i) Find Class & Object.

The concept of Class & objects is used to represent a class and all its objects. Class & object are considered important in the design because they are the least volatile, i.e over time, they tend to change the least compared to services of a class for instance. Coad and Yourdon suggest many ways to identify objects; from using the requirements specification to interviewing domain personnel to actually getting first hand experience of the problem domain.
ii) Identify Structures

By structures, Coad and Yourdon refer to the semantic concepts of generalisation/specialisation (genspec) and whole-part or aggregation in the semantic data model. Generalisation/specialisation structures give rise to inheritance hierarchies. Whole-part structures on the other hand give rise to associations between the part classes and the whole class.

iii) Identify Subjects.

Subjects are an interesting concept. It is built on the same idea as Clusters [Booch 86], and subsystems [Wirfs-Brock 90] and [Hendersen-sellers 90]. In this model, the top of an inheritance hierarchy or whole-part structure is promoted to the top to become a subject and represents the subsystem. Each such subject in the model subsumes its specialisations or parts. In large systems finding subjects early, assists in managing the complexity of the system.

iv) Define Attributes.

These are data types for which each object has its own value.

v) Define Services.

Services are the same as contracts or methods.

It is interesting to contrast this model with Wirfs-Brock's responsibility-driven approach. In this model, class & object is viewed as more stable over time than services. Emphasis is therefore placed on determining structure (class & and object) at the beginning of design. Services being volatile is left until last. Wirfs-Brock's approach on the other hand take the view that defining services at the beginning is important because it enhances encapsulation.
2.2.2.6 Summary of Object-oriented Designs (OOD) methods.

With the increasing popularity in the object-oriented paradigm and the development OOPLs, there is clearly a need to have OODs that can express object-oriented features of an application.

There is no standard OOD methodology as yet, however most authors agree on the basic steps required in the OOD process. Each of the OOD approaches discussed emphasised different phases of the software life cycle. Collectively their efforts span many phases of the software life cycle. This has made apparent the fact that in an OOD the phases are more blurred than in the traditional software life cycle.

Heavy investment in software systems derived from traditional systems analysis and design has obviated the need to have a mixed approach to design in which a traditional requirements analysis and design may be implemented in an OOPL and perhaps vice versa.

2.4 DISTRIBUTED MEMORY ARCHITECTURES

Distributed memory architectures are part of the wider field of parallel processing. In this section, a brief introduction is given about parallel processing and the general issues of parallel processing which are important to this project. Distributed memory architectures is then discussed in this context.

Perhaps two of the main factors the have influenced the rapid evolution and development of parallel processing are the quest for increased processing speed and the change in our perception of the way in which natural processes occur. It has long been realised that there is an upper limit to processing speeds of traditional sequential systems of the Von-Neumann architecture model, i.e current sequential systems, are restricted by the
physical nature of the hardware to an upper limit. Now, with the quest for increasing processing speeds, the solution forward is seen to lie in parallel processing architectures. Moreover most natural processes as in Realtime systems for instance occur in parallel and not in a forced sequential manner as happens in a sequential uniprocessor of the von-Neumann type. It appears logical therefore to develop parallel systems that reflect this natural parallelism. Capturing the parallelism in an application domain requires the development of both parallel software - parallel algorithms and architectures, and parallel hardware to support it. As will be seen, these components of parallel processing are very closely interrelated and imply that the development of a parallel software architecture must be done in the context of its corresponding parallel hardware environment.

2.4.1 Characteristics of parallel processing.

Two main characteristics of parallel processing are granularity and interprocess communication.

i) Granularity

This refers to the size of the unit of work that can be parallelised. On the one extreme we have the coarse grain level. At this level, the unit of work to be parallelised is normally the program module. In fine grain parallelism the statement is normally the unit of work to be parallelised. In between these two extremes, it is of course possible to have varying levels of granularity. Granularity level does affect the processing speed of an application. A fine grain level application can achieve faster speeds if it has good algorithms to divide up the parallel processing units, and interprocess communication is minimal. Otherwise it can be extremely inefficient. In practice, it is difficult to find good algorithms for fine grain applications and interprocess communication is on average high. Coarse grain applications on the other hand
have fewer interprocess communications and so can achieve faster processing speeds. It is clear here that to achieve optimum speeds and efficiency, there is a trade-off between the choice of granularity level and communication overheads.

**Kinds of Granularity**

DeCegama [DeCegama 89] identifies two basic kinds of granularity; event granularity and entity granularity. Event granularity refers to "the average amount of computation between two consecutive events of the same type", as for instance in the amount of computation between two consecutive message events, or synchronisation events. Entity granularity on the other hand refers to some program unit "with no part of it executable in parallel". Different instances of these two kinds of granularity may potentially exist in an application. Efficiently exploiting the parallelism inherent in such an application is one of the most difficult tasks in parallel processing.

**ii) Communication.**

Communication is essential in parallel systems. Two types of measure for communication can be identified. Throughput or bandwidth and Latency. Bandwidth is the rate at which communication ideally takes place in the parallel system, i.e without interruption. Latency on the other hand is the time it takes for a message to go from source to destination given that there is interruption in the communication path. Such interruption may include process idleness or contention due to switching or synchronisation events. To increase processing speed, it is obviously necessary to minimise latency especially in interprocess communication. This brings into focus important issues such as process allocation to minimise interprocess communication. These will be discuss in more detail later.
2.4.2 Architectures for Parallel Processing.

Parallel architectures have evolved in several directions. The broad categories of architectures are shown (fig. 2.5) [Decegama 89],[Shute 90],[Perrot 87]:

![Diagram of parallel architectures]

Figure 2.5. Broad classification of parallel architectures.

The three broad categories of architectures; the dataflow model, von-Neumann based models and reduction machine models [Fountain and Shute 90] represent three different strands of parallel architectures development. The Von-Neumann based models are further classified according to Flynn’s notation as SISD - Single Instruction Single Data stream, SIMD - Single Instruction Multiple Data stream; MISD - Multiple Instruction Single Data stream and MIMD - Multiple Instruction Multiple Data stream.

SISD machines are sequential machines that have only one processor and accept only a single instruction at a time. SIMD machine accept a single instruction but can broadcast this to many processors executing in parallel. Shared memory architectures belong to the SIMD model and we will not discuss them any further except in comparison with Distributed memory.
architectures. Distributed memory architectures which we are concerned with are of the MIMD type. Such systems consists of many processors each acting independently on its own instructions.

2.4.2.1 Issues in distributed memory architectures.

i) Memory management

Processes of a distributed memory architecture are said to be loosely coupled. A system of many such processes executes in parallel each with its own thread of control and operating on its own set of data. Processes communicate only by passing messages to each other.

ii) Communications network.

A system built upon this type of architecture has a communications network over which interprocessor communication takes place. Because of its parallel nature, internetwork control is essential in such a system. Such control can be achieved by using communications protocols, synchronisation mechanisms, routing techniques and so on. A parallel system's communications network has an impact on the system's capability, performance, optimum size and cost. Under a given network environment, we may for instance be interested in such issues as what the system can do, how efficient - in terms of processing speed it can perform, whether it is possible to scale up the size of the network and if so to what extend and at what cost.

iii) Partitioning and allocation.

Two important issues in distributed memory architectures are partitioning and allocation. Partitioning refers to the breaking up the application into logical units of processes or clusters of processes that can execute in parallel. Allocation is the subsequent mapping of
those units into processors according to the architecture design. It is often the case that an application has more logical process units than are processors. In such case some criteria must be used to fit processes into processors such that the resulting system has optimum performance.

iv) Fault tolerance.

Another important issue here is fault tolerance. The topology of a network will determine to a certain extent the reliability of the system. Depending on the application, one must choose a network topology that increases fault tolerance and hence maximise reliability.

2.4.2.2 Distributed memory architecture hardware and software.

The strong dependence between parallel hardware and software has already been mentioned. Such dependency requires that the development of software to capture the parallelism in an application domain should be done in the context of the target architecture. There are basically two approaches to capturing the parallelism in an application [DeCegama 89]. One approach is to write a program in a conventional language and then to let an intelligent compiler detect the parallelism inherent in the application. In the other approach, a parallel programming language is used and the programmer explicitly indicates using the syntax of the language, parts of the application that can be executed in parallel. Ideally one would prefer the former approach, however this is difficult. Current experience shows that it is better to have the programmer indicate as far as possible parallel parts of the application and then let the compiler restructure it for optimum execution. This brings in the general issue of software development environment for parallel systems. Parallelism brings with it an added dimension of complexity to the software support environment. This means that operating systems for parallel systems for instance should ideally be able to provide process scheduling
mechanisms, functionality to deal with load balancing, process creation and deletion, distributed file management and memory management and so on. In addition it is essential to have good debugging facilities and a good library of parallel algorithms.

2.4.2.3 Parallel Algorithms.

A parallel algorithm is one in which the computation steps can be divided among a number of processes. The design and implementation of good algorithms must be done with the target architecture in mind.

2.4.3 Summary of Distributed memory architectures.

In a distributed memory architecture, processes are loosely coupled and interact only via message passing. Two important characteristics of such parallel processing systems are granularity and communication. Capturing the parallelism inherent in an application is a difficult but important task. It requires the development of good algorithms. The implementation of such an application must be done in the context of the target architecture and should have a good software environment. It must also have a good communications network. The performance of the resulting system is a trade-off between the choice of granularity level and the communication overheads introduced as a consequence of the distribution of processes.
Object-oriented Parallel Systems.

This chapter surveys some of the research efforts that have gone into constructing parallel systems that utilise object-oriented ideas. As will be seen, most efforts have been directed into building Parallel Object-Oriented Languages (POOLs). A number of these languages will be discussed to highlight some significant implementation strategies used. But first, some general issues about OO parallel systems will be discussed.

3.1 ISSUES IN OBJECT-ORIENTED PARALLEL SYSTEMS (OOPS)

i) Granularity and Object Allocation.
Grain size is important in the development of OOPS. The language in which an OOPS is to be implemented must be able to express granularity. Objects are then constructed based on this granularity and form the basic unit of allocation. Object allocation refers to mapping objects to processors. In an ideal OOPS, objects would be allocated to processors in the network so that optimum performance is obtained. The allocation strategy used will
obviously depend on the nature of the application. Real-time systems for instance will impose timing constraints that will influence object allocation. However some main allocation objectives would be: i) minimisation of inter-process communication, ii) potential exploitation of parallelism, iii) optimum load balancing and iv) maximization of reliability of the system [Shatz and Wang 89]. Object allocation may be done statically or dynamically. Static allocation is a relatively easier task, however it may be too inflexible for some applications. Dynamic allocation although desirable in many application and improves performance, is nevertheless expensive to maintain.

**ii) Distribution transparency.**

Ideally, how objects are allocated and the actual physical mapping of those objects onto the network should be transparent to the user. He/she should not have to worry about the access paths and details of the communication channels to various objects throughout the network.

**iii) Inter-process communication.**

Objects in a distributed memory architecture must communicate and should do so in such a way the there is minimum message overhead. This as pointed out is dependent on the type of architecture used as well as on the object allocation strategy used.

**iv) Distribution of the inheritance hierarchy.**

It may turn out to be the case that an object allocation strategy would distribute objects of an inheritance hierarchy over many processors. In such a situation an inheritance problem arises. How does one object access an inherited method (of a super-class) that resides on a different processor? [Blair et al 91] noted some alternatives that have been tried:

a) restrict object instances of the hierarchy to reside on the same processor. The obvious disadvantage here is that it restricts object mobility.
b) "maintain a master copy of the hierarchy and cache slave copies of the required machines"

c) replicate the class hierarchy in some of the nodes. Both b) and c) have the disadvantage that each time changes are made, the replicas have to be updated in order to maintain consistency.

d) each instance of a class should have knowledge (pointers of some sort) of the location of all the classes whose methods it can inherit. In a distributed environment, such referencing mechanism would depend very much on the application.

3.2 SOME PARALLEL OBJECT-ORIENTED LANGUAGES (POOLS)

There are several alternatives for parallelising an object-oriented language. [America 87] identified the following:

a) Add the concept of process to a traditional language and have several processes execute concurrently or in parallel. This was basically how parallelism was added to Smalltalk-80. In this case however each process executes as if it were an ordinary sequential program. There is no integration and communication between processes is quite difficult.

b) Associate a process with every object. Objects execute in parallel but execution begins with one object only. An object is active only when invoked and message passing is synchronous.

c) Have objects as in b) but use asynchronous message passing. Queues are associated with each object to store messages and additional functionality is built into each object to manage asynchronous message passing activity. This is the approach
used in the ACTOR model [Hewitt 77], [Agha 86] and others as will be discussed further in the following sections.

d) Specify a body for each object which encapsulates its internal state. All objects becomes active and execute in parallel when the system is started and message passing is done explicitly and synchronously. This is the approach taken by POOL-T [America 87].

We discuss below some of the parallel object-oriented language models in the context of the above classification.

3.2.1 The ACTOR Model.

The ACTOR model for constructing POOLs was first put forward by Hewitt [Hewitt 77] and later developed by others including [Agha 86], [Agha and Hewitt 87], [Agha 90]. From this model, several other POOLs emerged notably ABCL [Yonezawa et al. 87], ACT++ [Kafura and Lee 90] and Act-1 [Lieberman 87].

The ACTOR model consists of five components: actors, a mail queue to store messages, messages, behaviours and acquaintances. Each actor is associated with a mail queue with a
unique address which is its identifier - equivalent to an object with a unique identity as shown

![Conceptual view of an actor](image)

**Figure 3.1 Conceptual view of an actor.**

Actors interact concurrently by passing messages to each other. The use of the mail queue allows for asynchronous message passing. An actor reads messages from the queue in FIFO manner and responds accordingly.

A message consists of the address of the caller actor, the name of the behaviour to be invoked and the required parameters. Behaviours and acquaintances - other actors which this actor can send messages to, allow an actor to assume different behaviours. The manner in which an actor responds to a message is determined by a behaviour script, which is basically a sequence of methods and acquaintances to be invoked. At any instance that the actor is active, it is executing a particular behaviour script. When completed, it picks the next message from the queue and executes its behaviour script. The fact that an actor can assume different behaviours depending on the message makes the actor model different from other traditional OOPLs where each object can only invoke methods defined in its class.

The ACTOR model supports side-effect free operations to ensure that there are no data dependent instructions.
3.2.2 POOL-T.

POOL-T is another parallel object-oriented language developed by Pierre America [America 86], [America 87], [America and Rutten 89] in an effort to apply object-oriented ideas in a multiprocessor environment to obtain parallel speed-up. The language was designed to be used in a network of up to 1000 processors.

3.2.2.1 Object oriented ideas of POOL-T

Objects and Classes
In POOL-T, every data item is an object. Part of the reason for this representation arises from the view that in a parallel processing environment, all objects - data structures as well as processes, should be active ".. carrying their processing capabilities with them.." as opposed to the idea that data are passive entities acted upon by processes. It is also claimed that such representation simplifies the language as well as give a unifying view. Objects are instances of classes and a class contains a description of the structure of a typical instance.

Units
A special organisational feature of POOL-T is the notion of a module; similar to the concept of modules in modular programming. Modules are referred to as units in POOL-T and are introduced to encapsulate related classes; a set of classes that interact with each other. It forms a coarse encapsulation structure around related classes and
provides further information hiding from other units. Three module types are defined: 
*specification units, implementation units* and *root units*. An implementation unit contains a set of class definitions and its corresponding specification unit "...describes which classes and which of their methods can be accessed by other units that use this unit." [America 78]. A root unit also contains classes and is one that can initiate execution of the system.

**Instantiation mechanism.**

Instantiation of objects is done by special procedures called *routines* which exist outside of a class. If a routine were in a class then an instance of that class would have to exist before it can be called - a chicken and egg problem. Routines may be called from anywhere in the system.

**Message passing.**

A POOL-T system consists of objects that interact via messages. Messages will be discussed further below (sect 3.2.2.2) but basically it is of the form

\[ d! m(p1,p2,...,pn) \]

where \( m \) is the method to be invoked, \( p1 \) to \( pn \) are parameters and \( d \) is the destination of the message.

**Inheritance.**

The concept of inheritance has not been incorporated into POOL-T. POOL-T was developed from a very strong theoretical basis. It was felt that the semantics of generalisation/specialisation on which inheritance is based are as yet unclear. The
specialisation of a subclass it is claimed, alters its behaviour and makes it difficult to formally reason about the relationship between the subclass and its superclass and hence difficult to verify the program. In parallel programming, such verification is much more important than in sequential programming.

3.2.2.2 Parallelism in POOL-T

Objects in POOL-T execute in parallel and communicate via message passing. Each object has a body that encapsulates its internal state. Objects become active upon execution of the system, each carrying out its own set of activities. This means that communication between objects must be done explicitly. In this case, synchronous message passing is used so that for each message sent by the client object of the form: \( d ! m(p1, ..., pn) \), there is a corresponding explicit answer-statement from the server object of the form \( ANSWER(m1, m2, ..., mn) \) where \( m1 \) to \( mn \) is a list of methods from which one will be selected to be executed. The use of synchronous message passing allows the programmer to have more control over the execution of the program.

3.2.3 POOL AND DOOM

POOL and DOOM [Annot and Haan 90] is another object-oriented approach to parallel computing. POOL is derived from the same family of languages as POOL-T described in section 3.2.2 and DOOM (Decentralised Object-oriented Machines) is a parallel architecture designed especially to support the execution of POOL programs.

3.2.3.1 Object-oriented ideas in POOL.
Objects, Classes and instantiation.

All data items are modelled as objects in POOL including integers and booleans and other scalar types. An object has an activity of its own and once created, it starts execution. Objects are the unit of parallelism. They execute in parallel but within each object, execution is strictly done in a sequential manner.

Classes form the template from which instances of the class can be created. As in POOL-T, classes are not objects themselves to which objects can send instantiation messages as in Smalltalk-80. Instead instantiation is done by special procedures called routines that exist outside of the class structure. A routine is associated with a particular class.

Units.

As well as classes, a POOL program consists of specification and implementation units similar to those in POOL-T (see section 3.2.2.1) and serve the same functions.

3.2.3.2 Parallelism in POOL and the DOOM architecture.

The DOOM architecture provides the environment to support Synchronous and Asynchronous communication of objects in a POOL application. It consists of a network of nodes which can be configured to any topology. A typical node consists of a data processor (DP) for executing object methods, a memory for storing data and code, a link to a host and a communication processor (CP) for passing messages. The role of the CP is interesting. The CPs of all nodes provide a separate network layer whose function is to route messages around the network. This relieves data processors from being directly involved in the communication process. Fig3 illustrates the CP communication layer.
3.2.4 TRANSPUTERS, OBJECTS AND OCCAM.

With increasing interest in Object-oriented parallel systems, some researchers have explored the possibility of harnessing the potential power of the transputer and Occam to support object-oriented ideas. This research direction has also been taken in this project as will be explained in chapter 4. In the following sections, we will discuss some of the significant issues that have arisen out of these research efforts.

3.2.4.1 The Parallel Object (PO) model.

[Ciampolini et al. 89] proposed a Parallel Objects model for building an object-oriented parallel system using Occam in a transputer network environment.

Object-oriented ideas in the PO model.
Objects

In the PO model, all objects are active and execute in parallel. An object encapsulates its internal state and hides it from other objects. It decides not only how but also when to respond to messages. The internal state of a typical object basically consists of a set of activities or methods, message queues, an internal state handler and a scheduler.

The invocation of methods is managed by the scheduler. The message queues consists of incoming messages which may be for this object or another object. A communications manager is associated with the message queues and manages the routing of messages to destination objects. The state handler is a process that manages all the object's internal variables. Objects in the PO model are assigned unique names which are defined globally.

Message Passing

Communication in the PO model is point-to-point and the propagation of messages through the network depends on the communication manager in each object. Messages contain the address of the target object as well as other parameters.

There is no notion of classification or inheritance in the model.

Parallelism in the PO model

Because Occam supports fine grain parallelism, both Inter-object and intra-object parallelism are possible. In the PO model, both parallelism modes are supported. Two modes of asynchronous communication are used. The pure asynchronous mode in which the client object does not wait for a response and a marked asynchronous mode in which the client may if necessary request a reply to a message. The asynchronous modes of communication are facilitated by the message queueing mechanism and the communication manager built into each object.
3.2.4.2 Fusing process and object.

[Thomas 89] proposed a model for building object-oriented parallel systems based on the fusion of the process and object paradigms in an Occam/Transputer environment. He first points out the following differences and similarities between Occam processes (or just processes from here on) and objects.

The main differences are that:

a) objects are not inherently concurrent whereas processes are and

b) instantiation in objects is a dynamic activity whereas in processes - at least in OCCAM processes instances are statically defined by the PAR construct.

Similarities include the fact that:

a) processes are composed of code (syntactic expression) and data while objects consist of class and instance;

b) objects use message passing while processes use communications.

With these characteristic features of objects and process, he proposed his model.

**Objects and classes**

A class in the model is a process that encapsulates a set of one or more processes which are methods of the class.

**Message passing / Communication.**

Tagged protocols are used in communication. A tag field in the message (protocol) identifies the method of the object to be invoked. Using tagged protocols, enforces information hiding. All messages are paired; there is always a response for every message sent.
Instantiation

Due to the static nature of OCCAM instantiation cannot obviously be done dynamically. To achieve this, Thomas introduced the idea of an Object manager. This is a large process whose functions are to i) create new classes and remove unneeded ones, ii) create class instances and delete unneeded ones and iii) reconfigure the connections between objects or classes as they are created or deleted. This last point is quite tricky because knowledge about the network connectivity has to be built into the object manager. In fact knowledge about instantiation and network connectivity is kept in a database which is encapsulated by the object manager along with other resources. Objects in the model perform instantiation or request instance/class deletion by making requests to the object manager.

Inheritance.

Thomas claims that inheritance is implemented by including the name of the superclasses in the class. During the instantiation process, correct connection channels to the superclasses are passed on to the instance.

3.2.4.3 Dynamic objects on Transputers

[Siet-Leng 91] developed further the idea of having 'manager objects/processes' such as proposed by Thomas discussed in the last section, to manage object queries. He proposed a parallel architecture in which there is a kernel consisting of manager objects, a kernel interface to multiplex messages between the managers and the user objects and the user objects themselves.

In the model, there are five managers: i) the object manager - which encapsulates important information about objects, ii) the execution manager - which responsible for configuring objects onto the transputer network, iii) the memory
managers - which deals with allocation and deallocation of memory that occur during instantiation and deletion of object instances, iv) the message manager - that facilitates object interaction and v) the kernel interface manager.

Managers in the kernel can interact with each other to satisfy requests from user objects. User objects on the other hand execute in parallel outside the kernel and only interact with it via the kernel interface.

One can see the advantages of having more than one manager process to deal with user objects requests.

3.2.5 Transputers and Parallel databases

A recent application of transputers to support OO concepts is in the development of a parallel database fourth generation language (P-DB4GL) [Gray 90a], [Gray 90b], [Gray 90c]. It is an attempt to modify a database fourth generation language DB4GL which already supported OO concepts so that it can operate on a transputer network environment. Initial findings have shown that the approach is feasible and there is potential for improvements.

3.3 Summary of Object-oriented parallel systems.

Features that are important in constructing Object-oriented Parallel systems include parallelism, the definition of object or process, allocation of objects among processors, inter-object/process communication and the distribution of an inheritance hierarchy over the communications network. Several approaches have been attempted to construct object-oriented parallel systems. The main approach has been to build POOLs. This led to the development of pure and hybrid POOLs. Pure POOLs of the ACTOR, POOL-T and POOL and DOOM models are large scale complete systems that are based on some theoretical model. They are generally large and quite expensive projects. Hybrid POOLs are an attempt
to incorporate parallelism into the framework of existing sequential OOPLs. It is a cheaper way to get a POOL but efforts to implement various features such as message passing and synchronization have often been restricted by the structure of the base language.

A more recent approach to build object-oriented parallel systems has been to harness the power of the transputer and use the Occam language. Even though there are restrictions in the Occam language with regard to implementing object-oriented features, there are ways to get around common problems. Certainly with the availability and low cost of transputers the advantages are there to pursue this approach.
An approach to exploiting parallel architectures to support Object-oriented ideas.

Object-oriented (OO) concepts and parallel architectures were discussed in chapter 2, and various applications of these concepts were reviewed. A review of some of the research that went into parallelising OO concepts followed in chapter 3. In the following sections, we present an analysis of the reviews which has helped in formulating the approach adopted in this project. An outline the approach will then be presented.

4.1 AN ANALYSIS OF OO APPLICATIONS.

4.1.1 OOPLs
As noted in chapter 2, the historical development of OO applications has been closely associated with OOPLs. This in our opinion made OO systems become synonymous with OOPLs - at least to those in the field of computing. In this context, utilizing OO concepts efficiently was seen as improving implementation strategies for various OO features within the framework of an OOPL. OOPLs can be classified into two broad categories; the pure
OOPLs e.g. the Smalltalk family, and the hybrid OOPLs e.g. C++, Objective-C and CLOS. The hybrid languages were the result of efforts to extend current traditional languages to include as far as possible desirable OO features. Because of the wide differences in the structures of base languages used to develop hybrid OOPLs, there is no uniform approach to implementing OO features. In fact most base languages either restrict or compromise the implementation of these features. Such restrictions have given rise to different interpretations of OO concepts and sometimes unnecessary confusion.

4.1.2 OOD METHODOLOGIES.

With the increasing popularity in the object-oriented paradigm and the development of OOPLs, there is clearly a need to have OODs that can express object-oriented features of an application. There is no standard OOD methodology as yet, and one of the reasons for this is that most of the OOD approaches that have been advanced were developed in the narrow context of some particular target development environment, e.g. Booch’s approach was developed with regard to Ada (an object based language) and Meyer’s approach was with regard to the development of the Eiffel System. Moreover each of the OOD approaches discussed emphasised different phases of the software life cycle. Collectively their efforts span many phases of the software life-cycle and has made apparent the fact that in an OOD, the different phases of the software life cycle are more blurred than in the traditional software life cycle. Heavy investment in software systems derived from traditional systems analysis and design is probably another reason for the different OOD approaches. Henderson-Sellers and Booch for instance advocate the need to have a mixed approach to design in which a traditional requirements analysis and design may be implemented in an OOPL and perhaps vice versa.

Despite these differences, there are it appears some clear steps which are basic in the OOD process.
4.1.3 DISTRIBUTED MEMORY ARCHITECTURES.

Two important characteristics of parallel processing systems and especially those based on distributed memory architectures, are granularity and communication. Capturing the parallelism inherent in an application requires the development of good algorithms. This is not an easy task. An important consideration in dealing with parallelism is to decide on the unit of parallelism - the grain size. As well as good algorithms, it is necessary to have a good language to express the parallelism, a good software development environment and of course a good target architecture. Communication issues become important when it comes to allocating processes to processors. Important considerations in communication include the number of inter-process links, the amount of inter-process communication that takes place, the mode of communication and the communication protocols involved. A good communications network is required and allocation of processes must be done in such a way that message overheads are minimised. The performance of a good parallel system is a trade-off between the choice of granularity level and the communication overheads introduced as a consequence of the distribution of processes.

4.1.4 OBJECT-ORIENTED PARALLEL SYSTEMS

Features that are important in constructing Object-oriented Parallel systems include the following:

i) extracting parallelism from an application - this includes defining the unit of parallelism and then seeing how this can be expressed,
ii) distributing units of parallelism among processes - this includes defining objects, seeing how objects or clusters of objects can be grouped into logical or desirable units and then mapping these units onto processors.

iii) designing a communications network that supports and controls inter-process communication.

Several approaches have been attempted to construct object-oriented parallel systems. As in the application of OO concepts, the first approaches have been to build POOLs. This led to the development of pure and hybrid POOLs. Pure POOLs of the ACTOR, POOL-T and POOL and DOOM models are large scale complete systems that are based on some theoretical model. They are generally quite expensive projects. Hybrid POOLs are an attempt to incorporate parallelism into the framework of existing sequential OOPLs. It is a cheaper way to build a POOL but efforts to implement various OO features such as message passing and synchronization have often been restricted by the structure of the base language.

A more recent approach to build object-oriented parallel systems has been to harness the power of the transputer and use the Occam language. Even though there are restrictions in the Occam language with regard to implementing object-oriented features, there are ways to get around common problems. Certainly with the availability and low cost of transputers the advantages are there to pursue this approach.

4.2 THE RESEARCH PROBLEM AS SEEN NOW.

The benefits of OO concepts and distributed memory architectures have already been made clear. We have seen that approaches to utilise OO ideas in sequential and parallel environments have been various with each approach having its own advantages and disadvantages. This has opened up the possibility of using mixed approaches to obtain
Object-oriented parallel systems. We show in fig. 4.1 how these different possibilities which are seen in the context of Object-oriented development and the development of parallel systems can be pursued.

Figure 4.1 The possible approaches to developing Object-oriented Parallel systems.

Whether Parallel Systems Analysis and Parallel Systems design are real possibilities is yet to be discovered. The trend in traditional systems had been from programming languages to structured design and then to structured analysis. Similarly Object-orient systems started with programming languages. From this sprang object-oriented designs and now according to Coad and Yourdon we have Object-oriented analysis. There is reason therefore to believe that parallel systems will evolve in the same direction. Regardless of the existence of such parallel
systems development methodologies, we know that processes exist in the problem domain of a parallel application and that in the process of capturing the parallelism in an application we are actually creating an abstract model of the application in which we have processes in the solution space.

From fig. 4.1 we could envisage a design methodology in which we capture the object-oriented features as well as parallel features of an application - transitions 2 and 3 in fig. 4.1 to obtain an abstract model of an object-oriented parallel system consisting of objects and or processes in the solution space. Such a model can then be implemented using a POOL - transition 7, or a parallel language - transition 8, or possibly an object-oriented language or object-based language such as Ada - transition 6.

In the light of preceding discussions on OO concepts and parallel systems it was decide to take the following approach in this project: To use an object-oriented design methodology to capture object-oriented features and inherent parallelism of a model application - using transitions 2 and 3 in the above diagram and then to use the facilities provided by the transputer and Occam - transition 8 to implement an object-oriented parallel system which would reflect the nature of the application. An important aim is to investigate the design translation of the model.

It is a mixed approach in which we implement an extended object-oriented design in a non-object-oriented language Occam.

4.3 AN OUTLINE OF THE APPROACH.

We outline below the proposed approach to constructing an object-oriented parallel system.
i) Select a suitable application model; one which has some object-oriented features as well as parallelism in it.

ii) Use an OOD methodology to capture the OO features and parallel features of the model.

iii) Identify ways to represent OO features and parallel features in the Occam and transputer environment.

iv) Implement the model in a distributed memory architecture environment.

v) Evaluate the performance of the application.

vi) Make generalisation about the design translation from OOD to implementation.
5.1 INTRODUCTION

Our view, of an Object-oriented parallel system is one of a model of a real life system with inherent object-oriented features and parallel features. Such a model consists of a collection of objects distributed throughout processors in some network according to a target distributed memory architecture. Objects form the basic unit of allocation among processors. Objects encapsulate their own services or methods and data and use the services of other objects only by interacting with them using well defined communication protocols. Inter-object communication is controlled by a communications network.

Clearly a design method to model such an Object-oriented Parallel System should be able to capture both OO features as well as parallel features of the problem domain. These features have already been discussed in the last three chapters. Here we list some of the features that will form the basis of the design.
Important design features of a distributed memory architecture type parallel system:

Granularity

Inter-process communication

Process allocation

Important design features of an object-oriented system:

Objects and classes

Object services

Object relationships and interaction

clusters, subsystems and inheritance hierarchies.

5.2 THE DESIGN APPROACH.

The design approach that is developed here is based on work reviewed in chapter 2; [Bailin 89], [Wirfs-Brock 90], [Henderson-Sellers 90] and [Coad and Yourdon 90]. It is basically an Object-oriented design with modifications to accommodate parallel features of an application. There are five main phases in the design process and these are outlined below:

I. Find objects and identify their services

II. Construct an object interaction diagram from the information in phase I and enhance it.

III. Define Class services.

IV. Map design to target implementation domain.

V. Identify clusters or subsystems for allocation to processors.
The first phase of finding objects and identifying services follows closely Bailin's object-oriented structured analysis type approach [Bailin 89]. In this process, an Information Flow diagram (IFD) consisting of active entities is constructed from an initial entity-relationship diagram (ERD). Entities undergo an iterative decomposition process in which new entities found below existing active entities are added to expand the IFD. In this process, entity services are also identified. The DFD-Edit [IDDK tools 89a] modelling tool is used in this process.

The second phase consists of constructing an Object Interaction Diagram of the Extended IFD derived in phase I and then enhancing it. Enhancements include adding the hypersemantic concepts of generalisation and aggregation along the line of Coad and Yourdon's genspec and part-of structures [Coad and Yourdon 90] as well as the type of relationships between entities.

The third phase consists of defining clearly the services of objects which were identified in phase I. The method follows closely Wirfs-Brock's idea of defining object responsibilities [Wirfs-Brock 90].

Up to the third phase, the design is independent of target implementation domain. The fourth phase maps the design to the target implementation domain. In this case we have chosen the Transputer/Occam environment but it could well be any other implementation domain. The resulting model has been called a domain specific logical design (DSLD).
The last phase consists of partitioning the model derived in phase 4 into logical units. Such units will serve as the basis for allocation of object or object clusters to processors. This phase has been left until last because the mapping of a design to the target implementation domain may impose restrictions on the way in which object allocation can be done.

In the following sections, each of the above phases of the design process will be described in more detail. But first we describe the problem domain that has been used to illustrate the ideas put forward in this project.

5.3 THE PROBLEM DOMAIN.

The objective here is to use a model which is simple but which contains important OO as well parallel features. It was decided to use the example of an Airline Reservation system as the model for the design and implementation of an object-oriented parallel application. This example has been used by others [Luqi 91] working on these kind of problems. An airline reservation system is a real time system which allows many travel agents in a distributed environment to interact concurrently by making travel arrangements for passengers. It can be classified as a data intensive application with plenty of message passing. There are basic entities in such a systems which are identifiable. For instance it has travel agents, airline manager(s), flights, reservations, passengers, tickets, fares and so on.
5.4 PHASE I. FINDING OBJECTS AND IDENTIFYING SERVICES.

The design process assumes that a requirements definition of the system is available. This document contains the user's perception of what the system should look like and what it should do. This is used as input to the first stage of the design. Suppose that the following simplistic requirements definition of an airline reservation has been submitted.

**Design an airline reservations system which will help travel agents sell tickets to passengers on commercial airlines.** The system should be able to handle up to 300 travel agents. These agents would be distributed over a wide geographic region. The system should be fast enough not to annoy travel agents. Agents may make reservations, cancel reservations, access their own reservations, as well as have limited access to other facilities of the system's database. The system will handle flights from several airlines. Managers of different airlines should be able to add their own flights, delete flights and add details of fares. An airline manager is not allowed to change another airline's flight details. The system should be able to cope with a volume of about 30,000 passengers per year.

From this requirement definition we set about finding objects and identifying their services according to the following steps:

i) Identify entities from the requirements definition and construct an *initial entity-relationship diagram (ERD)*.

ii) Identify **active entities** and construct an *information flow diagram (IFD)* using the active entities.

iii) Decompose each active entity to see if more entities exist below. In this decomposition process, identify the services of each entity.
iv) Bring each new found entity to the top level and expand the IFD.

v) Repeat steps ii) and iv) until the IFD is deemed complete. The result is an expanded IFD.

5.4.1 The initial ERD and IFD. (steps i. and ii.)

We begin the design process by identifying possible entities in the system. Such entities will later on become objects of the system. We call them entities and not classes at this stage because by our definition classes represent entities which encapsulate their own data and behaviour. These have yet to be identified.

A common method that has been used to find entities is to look for nouns in the requirements definition. Such nouns most often represent entities of interest in the system. In the model requirements definition above, the nouns have been highlighted in bold. From this set of entities, an initial entity relationship diagram (ERD) as shown below (fig 5.1).

![Fig.5.1. A initial ERD of the airline reservation system model](image)
This is a conceptual representation of the system which has been modelled using ERA-Edit (IDDK 89b). The rectangles represent entities and the ovals represent relationships between the entities. At this stage the relationships indicate that there is some interaction between the connecting entities. The nature of these relationships will become clear later on in the design. There are three entities highlighted in the requirements definition that have not been included in this diagram: flight, reservations and fares. As it turns out, they are conceptual entities that are part of the AirlineReservationSystem entity. They will be revealed in the next stage of the design.

From the above diagram we identify active entities. These are entities which are important to us at this stage in the design. They would be entities for which we can identify services that are of interest. From fig. 5.1, the following entities are identified as being active; AirlineReservationSystem, AirlineManager, TravelAgent. The entities tickets and passengers and airline are physical entities which are considered external to the system we want to model and so have been omitted. An information flow diagram (IFD) is then constructed using DFD-Edit (IDDK 89a) from the active entities identified. IFDs are similar to Data flow diagrams (DFDs) of the DFD-Edit modelling tool and features have similar meanings. An IFD of fig. 5.1 is shown in fig. 5.2. The bubbles of an IFD represent active entities that encapsulate behaviour, data and possibly other entities. They are named with brackets to distinguish them from data flows. The labelled arrows connecting bubbles represent the flow of information or messages between entities. It is possible to have external entities in and IFD representing the flowing of information to and from the model.
5.4.2 Decomposition of active entities - (step iii)

The next step in the design is to decompose active entities in the IFD (fig.5.2.) to see if further entities exist below. In the decomposition process, entity services will be identified. If a new entity is found, it will be brought up to the top level of the IFD and added to it. Services will then be identified and allocated to the new found entity. This may mean moving some services from some of the existing entities to the new one. The first level decomposition of entity ARSystem is shown in figure 5.3.
Figure 5.3. Decomposition of entity ARSystem

The bubbles in the diagram indicate the processes or functions performed by the entity ARSystem. The open rectangles represent data stores and as the name implies, data stores
are data repositories. The arrows indicate information flowing into or out of a process. There are also channels which are indicated by numbered circles which are used for inter-entity communication. The number in a channel indicates an entity that can communicate with this entity. From figure 5.2 we see that the entity numbers 2 and 3 refer to the entities Airline manager and TravelAgent respectively. By our definition, a function that performs a service for another entity is a service or method of this entity. The names service and method will used interchangeably in this project. Thus according to the notation used in our diagram, all bubbles that connect to channels are identified as services of the entity at this level of decomposition. Services can of course be decomposed to lower levels. Such decomposition basically means splitting a service into two or more finer grained component services and private functions at successively lower levels. From figure 5.3, we note that all bubbles are services of ARSystem at this level of decomposition.

Possible entities are found by looking for clusters of processes and/or data stores that suggest separate entities. We notice from fig.5.3, three clusters; one centred around the data store Reservation, one around Flight and one around Fares. We note from the original requirements definition (section 5.4) the entities identified by these names. These three process-data store clusters are clearly entities which are separate from ARSystem and should be removed from under ARSystem and added to the IFD in fig.5.2. Services are then allocated to them as shown below (tab. 5.1).

<table>
<thead>
<tr>
<th>Entity</th>
<th>Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reservation</td>
<td>Reserve, Cancel, List Agent reservations</td>
</tr>
<tr>
<td>Flight</td>
<td>Find flight, Add flight, Drop flight, Display flight,</td>
</tr>
<tr>
<td>Fares</td>
<td>Add fare, display fare.</td>
</tr>
</tbody>
</table>

Table 5.1 New entities found under ARSystem and their services.
It has been mentioned that a service can be further decomposed into finer lower level services and private functions - those that are not directly linked to an outside entity. While the level of decomposition depends on the designer, it is important that all services regardless of the decomposition level at which they are defined should be recorded explicitly as such. This should be done so that other entities can identify the correct service when making a service request to this entity. Consider the entity *Reservation* (tab. 5.1). If we decompose the service *Reserve*, we get the following diagram (fig. 5.5).

![Diagram of Service Reserve Decomposition]

**Figure 5.4. Decomposition of service Reserve.**
At this level of decomposition, Reserve is defined in terms of two lower level services - Check flight and Check booking and two private functions - Check Passenger Reservations and Make Reservation. If the designer so decides, he might choose Check flight and Check booking as services rather than Reserve. Otherwise communication between the services and other entities will have to be done via Reserve. Note from fig. 5.4 also that the two private functions use a data repository ReservationDetails. Decomposition thus assists in identifying further entities, services as well as private data used by an entity. The decomposition of the service Cancel (another of Reservation's services) reveals the same pattern as for Reserve as shown (fig. 5.5).

![Diagram](image)

**Figure 5.5. Decomposition of Service Cancel**

If we proceed in the same manner to decompose the services of entities Flight and Fares, their services at lower levels will be more fine grained and specific and more detail about the data types used will be revealed. Details of these are given in appendix A.
Going back up to the top level IFD (fig. 5.2), we decompose the entities *AirlineManager* and *TravelAgent* in turn. The first level decomposition of *AirlineManager* is shown in fig. 5.6.

![Diagram of the first level decomposition of *AirlineManager*](image)

Figure 5.6. *First level decomposition of entity AirlineManager*

The entity has one service *Interpret Manager Query* and two private functions *Get Manager query* and *Parse Manager Query*. We can identify two clusters which could be possible entities; one around the data store *ManagerQueries* and the other around the data store *ManagerQueryFormats* which indicate two possible entities. We shall call them *ManagerQuery* and *ManagerQueryFormat* respectively, remove them from under the entity *AirlineManager* and bring them up to the top to further expanding the IFD.

64
The new entities and their services are shown below (tab. 5.2):

<table>
<thead>
<tr>
<th>Entity</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>ManagerQuery</td>
<td>Get Manager Query</td>
</tr>
<tr>
<td>ManagerQueryFormat</td>
<td>Parse Manager Query</td>
</tr>
</tbody>
</table>

Table 5.2 *New entities extracted from under the entity AirlineManager.*

If we decompose the entity TravelAgent we find that it is virtually similar in structure to AirlineManager and as in Airline manager, we identify two new entities which can be removed and brought up to the top to expand the IFD. The new entities are shown here with their services (tab 5.3).

<table>
<thead>
<tr>
<th>Entity</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>TravelAgentQuery</td>
<td>Get Travel Agent Query</td>
</tr>
<tr>
<td>TravelAgentQueryFormats</td>
<td>Parse Travel Agent Query</td>
</tr>
</tbody>
</table>

Table 5.3 *New entities extracted from under entity TravelAgent.*

5.4.3 Expanding the IFD. (step iv)

The process of identifying services and new entities is iterative and would continue until the designer is satisfied with the expansion. The model used here has been greatly simplified to illustrate the design process, however in a real system the task will obviously be more elaborate. Details of all the entities and services identified in the airline reservation model are in appendix A. The extended IFD derived from information accumulated in the last three steps
is shown (fig.5.7) and will be the input to the second phase of the design process - constructing an object interaction diagram (OID).

Figure 5.7 An extended Information Flow diagram (IFD) of the Airline Reservation system model.
5.5 PHASE II. CONSTRUCT AN OBJECT INTERACTION DIAGRAM AND ENHANCE IT.

Using the extended IFD (fig.5.7), an Object Interaction diagram (OID) is constructed. Entities are now called classes as their services and data have already been identified. A first cut OID of the model constructed using the ERA-Edit (IDDK 89b) modelling tool is shown in fig.5.8. Note that entity ARSystem in fig.5.7 has been split into two classes; Am_ARSysInterface and Tr_ARSysInterface to reduce the effect of a possible bottleneck in the interaction between AirlineManager and TravelAgent and the classes Flights, Reservation and Fares. In object-oriented parallel systems having such bottlenecks can lead to contention problems and reduced performance.
The OID model shown here is very similar to the ERA semantic data model; it is a flat two dimensional system of classes and relations. The rectangles represent classes while ovals represent relationships between objects. The difference however is that classes encapsulate services as well as data and the relationships at this stage represent the fact that there is some interaction between objects. Relationships between objects are important especially in the context of an object-oriented parallel system where it represents inter-object communication over some network. It is therefore important to determine the nature of these relationships. Some similarities can be drawn between OID relationships and ERA relationships of semantic...
models. A comparison based on the following relationship types [Maciaszek 89], is made between the two models; degree, membership, regularity, connectivity, generic relationships and aggregate relationships.

<table>
<thead>
<tr>
<th>Relationship type</th>
<th>ERA model</th>
<th>OID model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Degree</td>
<td>unary, binary, ternary</td>
<td>binary, ternary.</td>
</tr>
<tr>
<td>Membership</td>
<td>partial, total</td>
<td>must be total</td>
</tr>
<tr>
<td>Connectivity</td>
<td>1:1, 1:N, N:1, M:N</td>
<td>1:1, 1:N, N:1, M:N</td>
</tr>
<tr>
<td>Regularity</td>
<td>weak, regular</td>
<td>weak, regular</td>
</tr>
<tr>
<td>Generalisation</td>
<td>generic relationships</td>
<td>generic relationships</td>
</tr>
<tr>
<td>Aggregation</td>
<td>aggregate relationships</td>
<td>aggregate relationships.</td>
</tr>
</tbody>
</table>

It can be seen, that there is a close correspondence between relationships types in the two models. There are however semantic differences and these will become more apparent when the design is mapped to its target implementation domain. Membership in and OID for instance must be total because all instances of a class must have the same behaviour. Regular relationships imply support classes or processes. In a Transputer/Occam environment regularity and connectivity are closely related because the nature of communication in Occam is such that, relationships between 1:N or N:1 and M:N classes must support inter-object communication. Generic relationships are introduced to represent possible inheritance hierarchies in the system. With these possible relationships, we proceed to enhance fig.5.8. The enhanced OID is shown in fig. 5.9.
Fig. 5.9. Enhanced Object interaction diagram.
Enhancements to the first-cut OID include the following:

i) Generic and aggregate relationships

Four generic relationships have been identified. The first two represented by superclasses *Query* and *Gparser* have been identified in a bottom up manner, i.e., by generalisation. In this case it is clear that *ManagerQuery* and *TravelAgentQuery* are types of queries. We therefore create an abstract generic type *Query* and make it the superclass of these two classes. Similarly *ManagerQueryParser* and *TravelAgentQueryParser* are both types of parsers. So a superclass *Gparser* is create to generalise the two types.

The other two generic relationships represented by the superclasses *Flights* and *Fares* were created in a top down manner, i.e., specialisation. In the case of *Fares* we determine if it has any specialisations which are significant in the system; we identify subtypes *SpecialFares* and *StandardFares* in this process. The same applies for *Flights*. Possible aggregate relationships have been left out in this model for simplicity.

ii) Connectivity and regularity.

The original requirements definition stated that the airline reservation system should be able to cope with up to 300 travel agents and use several airlines. This means having multiple instances of the classes *TravelAgent* and *AirlineManager*. A class with multiple instances is indicated on the OID by a semi circle at the end of a connection. With multiple instances we get 1:N relationships as between *ManagerQuery* and *AirlineManager* or *Tr_ARSys_INTERFACE* and *TravelAgent* and M:N relationships as between *AirlineManager* and *TravelAgent*. It has already been mentioned that in the context of the target Transputer/Occam implementation environment regularity and connectivity are closely related. Regular relationships on the OID are indicated by shaded ovals. Notice that they connect 1:N, N:1 and M:N
classes. Ternary relationships such as r7, r8, r9 are also regular because they play an important role in inter-object communication.

5.6 PHASE III. DEFINE CLASS SERVICES.

As Wirfs-Brock et al [Wirfs-brock 89] have pointed out, it is very important to identify class services or responsibilities clearly as early as possible in the design so that communication protocols can be defined. Services are defined in the context of a client-server model in which objects are seen as either servers providing services to other objects or clients requiring the services of other objects. In this model, it is also possible for an object to be both a client and a server. The number of clients and/or servers that a class has is the class’s visibility range. The set of all services provided by an object constitutes its interface. Other objects can see this particular object only through this interface. Most of the class services were identified in the decomposition process in phase I, however enhancements made to the OID (fig.5.9), have added some new classes. Services for these will have to be identified either by regrouping current services under other classes or defining new ones. Class services are defined in a class description format similar to that given in [Wirfs-Brock 90] as shown below:

**Class description format.**

Class : <class name>
Superclasses : [List of superclasses of this class]
Subclasses : [List of Subclasses of this class]
Clients : [List of classes that serve this class]
Servers : [List of classes that this class serves]
Services : [List of services that this class provides for other classes]
Private functions : [List of all private functions.]
Using this format, we can define the following classes:

Class : Gparser.
Superclasses : None
Subclasses : ManagerQueryParser, TravelAgentQueryParser
Clients : None
Servers : provides shared services for ManagerQueryParser, TravelAgentQueryParser
Services : ParseFindFlight, ParseNumeric.
Private functions : None.

The above is an example of a superclass formed by generalisation. The two services ParseFindFlight and ParseNumeric that were previously duplicated in classes ManagerQueryParser and TravelAgentQueryParser have been removed and regrouped under Gparser. [See fig 5.9].

The following three classes form the Flights generic relationship. The subclasses OneStopflight and MultiStopFlight have been derived by specialisation.

Class : Flight
Superclasses : None
Subclasses : OneStopflight, MultiStopFlight
Clients : None
Servers : Fares, Reservation.
Services : Findflight, Addflight, DropFlight, DisplayFlight.
Private functions : None.

Class : MultiStopFlight
Superclasses : Flights
Subclasses : None
Clients : Fares.
Servers : Am_ARSyst_Inteface, Tr_ARSyst_Inteface
Services : AddlegtoSchedule, DroplegfromSchedule +
           inherited services Findflight, Addflight, DropFlight,
           DisplayFlight.
Private functions : None.
Class : OneStopFlight
Superclasses : Flights
Subclasses : None
Clients : Fares.
Servers : Am_ARSys_intf, Tr_ARSys_intf
Services : Inherits all services from Flight
Private functions : None.

In the above case, specialisation was achieved by adding new services to MultistopFlight.

Notice that OneStopFlight inherits all its services from Flights. The rest of the classes are described in the same manner and have been moved to Appendix B.

5.7 PHASE IV. MAP DESIGN TO TARGET IMPLEMENTATION DOMAIN.

Up to this stage, the design has been general and independent of implementation domain. Mapping the design to the target implementation domain requires a clear understanding of particular characteristics of the domain that will restrict or enhance the mapping process. The result of this mapping is a design which is specific to that domain and we have called it a Domain Specific Logical Design (DSLD). In the next section we discuss features of the target implementation domain: the Transputer/Occam environment, that will affect the mapping process.
5.7.1 Occam and the transputer. (Characteristics affecting the design)

The Occam language is a parallel non-object language that was designed to be used on the transputer. Grain size of parallelism can be defined at different levels in Occam; at statement level which is as fine grained as one can get, at block level using constructs such as PAR, ALT and SEQ and at procedure level. The way objects have been defined in this design makes the Occam procedure the most logical unit for parallel distribution. This will be discussed in greater detail in chapter 6. Occam uses synchronous communication via unidirectional channels and inter-process communication is point-to-point. This implies that pairs of channels must be used for bi-directional communication and that some routing mechanism must be used to pass messages to destinations that are beyond adjacent neighbours. However point-to-point communication minimises contention problems and increases communication bandwidth.

The network of nodes in a transputer can be of any arbitrary size, however each node contains only four physical links. Each link makes available two inter-node Occam channels; one each way which means that each node can have at most 8 Occam channels - 4 in and 4 out. Within the Occam programming environment, physical links are transparent to the programmer.

5.7.2 The Domain Specific Logical Design

Taking into account the restrictions imposed by the Transputer/Occam environment, we come up with the following mapping (fig. 5.10).
Figure 5.10 A Transputer/Occam mapping of the airline reservation system design

A closer look at the Domain specific logical design (fig. 5.10) will reveal that the general structure of the design is basically the same as before the mapping (fig. 5.9). The main changes have been with regard to the relationships between classes. With the transition, the relationships have taken on new meaning. The regular 1:N/N:1 relationships r1, r2, r3, 4, r5 and r6 in fig. 5.9 have become 1:N/N:1 multiplexors/demultiplexors. Similarly the regular M:N relationship r13 has been transformed to an M:N multiplexor and regular ternary relationships r7, r8, r9 have been transformed to routers. Multiplexors, demultiplexors and routers are needed in the Transputer/Occam environment if instance-to-instance communication is required. Although they are not considered as genuine classes that provide services, in an object-oriented parallel system they play an important supportive role in inter-object communication. They have thus been called support objects.

Two other support objects that have appeared in the transition are Am_objman and Tr_objman. These are object managers of the type discussed in [Thomas 89] and [Siet-Leng 91]. They have been introduced to manage instances of classes that have multiple instances; in this case Amanager and Tragent. It is a way of getting round the static nature of the Occam language. The structure of object managers will be discussed further in chapter 6.

It is important to note that changes in the mapping process have been mainly with regard to relationships. It makes clear the fact that relationships (which imply inter-object communication) are an important issue in the mapping of a such a design as this to a parallel architecture environment.
5.8 PHASE V. IDENTIFY UNITS OF DISTRIBUTION.

(Identify classes, clusters or subsystems for allocation to processors)

5.8.1 Introduction

An obvious consideration in distributing processes to processors is the choice of the grain size for distribution - onto a target transputer network. At fine grain level, we could distribute all Occam parallel statements. At the other extreme we could distribute clusters of parallel processes. The choice of grain size will depend to a some extent on the type of application for which the design was made. In any case whatever the choice of granularity may be, allocation of units must be done in such a way that the following allocation objectives are met:

i) inter-process communication is minimized

ii) potential exploitation of parallelism is maximised

iii) optimum load balancing is maintained and that

iv) overall, the reliability of the system is maximised.

In an object-oriented parallel system, the smallest logical unit for distribution is the class which by our definition encapsulates services as well as data.
5.8.2 Allocating Classes. (The ideal case)

In an ideal situation we would have one processor for every class so we could envisage each of the classes in fig.5.10 residing on a separate processor - 19 processors in all. In such a situation, classes with multiple instances such as Tragent and Amanager will have their instance run on the same processor. Because of the limit in the number of physical links on a transputer node, 1:N multiplexors and N:1 demultiplexors will have to reside on the processor that has the class with multiple instances. Consider for instance the class Tragent and its multiplexors (fig. 5.11).

![Diagram of Tragent class with multiplexors](image)

**Figure 5.11 Allocation of Tragent - a class with multiple instances.**

All the multiplexors TrQmux, Tr.objmux and Trparser reside on the same processor as Tragent (represented by instances tr1, tr2, tr3). Incoming messages from TrQuery for
instance are multiplexed and sent to instances $tr1, tr2$ or $tr3$ while outgoing messages are demultiplexed and sent out.

M:N multiplexors present a difficult problem in the allocation process because the number of instances of a class at any instant cannot be predetermined. In this case, instance to instance communication as for instance between an instance of TravelAgent and an instance of Amanager will be difficult to implement in an environment which requires the static allocation of communication channels. There are however strategies for getting round such problems. With routers there are many channels however these are fixed and can be predetermined.

5.8.3 Allocating clusters of classes.

In reality, there is not always a processor available for every class. If there are more classes than processors - as is normally the case in large systems, then classes will obviously have to be organised into logical or desirable clusters and then distributed. Clustering criteria must take into account the allocation objectives (section 5.7.1). Assuming that the problem being modelled does not impose constraints on the way cluster distribution should occur, the most logical way to cluster classes would be based on the occurrence of natural clusters or subsystems such as i) generic relationships ii) aggregate relationships and iii) groups of classes closely related by their interconnections and functionality. Such subsystems can in turn be grouped to form coarser grain clusters of subsystems at a higher level. The desired level of clustering will be determined by the number of available processors. For our design (fig. 5.11) a possible first level clustering is shown if figure 5.12.
In the above diagram, the generic relationships are represented by the subsystems *Query*, *Parser*, *Flight* and *Fares*. *Amntface* and *Trntface* have been merged to *ARS_Jntfce*. We can thus distribute the system into 8 processors. Taking into consideration the fact that there is strong interaction among clusters *ARS_Jntfce*, *Reservn*, *Fares* and *Flight* we obtain the following higher level cluster.

Figure 5.12. *First level clustering of domain specific logical design.*
The design at this level of clustering can now be distributed among 5 processors. Once clustering has been done as above, the rest is a matter of implementation.
6.1 INTRODUCTION.

Part of the aim of this project was to implement the design derived in chapter 5 using a non-object-oriented language in a distributed memory architecture environment - option 8 of fig. 4.1 in chapter 4. As has already been mentioned, such a target implementation environment is provided by the Transputer and Occam; Occam is a non-object-oriented parallel language and the transputer provides the distributed memory architecture. Work reviewed in chapter 3; [Ciampolini et al 89], [Thomas 89] and Siet-Leng 91] have shown that there are possible ways to take advantage of Occam and the transputer to support object-oriented ideas. We mention again that the main benefits of the Transputer and Occam are that they are widely available, cheap and also they provide a parallel architecture environment that is rapidly becoming more advanced. In this chapter, we give a brief description of the Transputer and Occam, how OO concepts have been represented in Occam and finally how they were used in the transputer environment to implement the design derived in chapter 5.
6.2 THE TARGET LANGUAGE - OCCAM

Without going into the history of the language, Occam is essentially a language based on work done by Hoare, [Hoare 78], [Hoare 85] on the CSP (Communicating Sequential Processes) language - a mathematical based model for specifying the behaviour of concurrent processes. The main objective in the design of Occam was to keep it simple while providing sufficient features for programming distributed systems. The language was developed in close association with the transputer environment and was intended to be the lowest level language normally used for programming transputers. Some important features of Occam are described below [Burns 88], [Wexler 89], [Kerridge 87].

6.2.1 Occam Processes

Occam supports parallelism at different levels resulting in a hierarchy of process levels.

i) At fine grain level we can have primitive processes such as the input process represented by '?', output process represented by '!', the STOP and SKIP processes.

ii) At the next level up, we have blocks and channels and then

iii) constructs such the SEQ, PAR, ALT, PRI PAR, PRI ALT with which one can construct processes that can execute in parallel.

iv) At the highest level we have Occam procedures which can encapsulate further procedures, functions and processes.
All Occam processes are unique but are unnamed and processes executing in parallel must be explicitly indicated by the use of the PAR construct in a program.

6.2.2 Communicating in Occam

All Occam processes communicate by sending messages over channels using the CSP model of synchronization [Hoare 78]. Channels are unidirectional which means that bidirectional communication between two processes must use at least two separate channels. A channel is normally declared as a data type; an integer channel for instance will only allow integer data through it. Groups of data can be sent simultaneously through special channel types called protocols. These will be discussed further later.

Physical links connecting processors are defined as channels in an Occam program in the same way as internal channels. This uniformity relieves the programmer from the need to consider physical inter-process communication in the implementation process until the configuration stage.

6.2.3 Process Allocation.

The logic of an Occam program is independent of the target transputer architecture. In fact it is possible and indeed desirable to design, implement and test an Occam application on a single transputer before distributing it through the network; if it works for one transputer then the program works and it should work for any configuration with minimum adjustments. Allocation thus comes as the last stage in the development of an application and is referred to in Occam as Configuration.
6.3 THE TARGET HARDWARE ENVIRONMENT -
THE TRANSPUTER

In this section, we give a brief explanation of the transputer in the context of the Transputer/Occam environment. Detailed information about transputers - the IMST800 model is shown in appendix D.

The transputer is a VLSI chip designed in close association with Occam to support concurrency and synchronization. A typical transputer node consists of a microprocessor, a limited amount of local memory - normally 4KBytes, 4 high speed inter-node links, an interface for system services, an interface for off-chip memory and in the case of the IMST800 series, a floating point unit. In a multi-transputer network, each node is an autonomous unit that has control over its local memory and off-chip memory and so does not have to compete with other nodes for memory or even data; contention problems therefore do not arise. Each transputer link has a very high data transfer rate - in the order of megabytes/sec with automatic synchronization in each direction. Such high data transfer rate is made possible by the use of DMA block transfer mechanisms that do not directly involve the attention of the microprocessor. It thus relieves the microprocessor to attend to other tasks and means that processing and communication can take place simultaneously.

As has already been mentioned, some system services have been microcoded into the transputer hardware to facilitate processing. These include scheduling, the support of two queues - a high and low priority queue, synchronization, process suspension and context switching. These built-in facilities provide an efficient "Occam engine" to support the efficient execution of Occam applications.
Apart from the fact that Occam is a non-object-oriented parallel language, it also has a static nature and lacks many of the data types normally found in many hybrid OO languages to support OO concepts. It for instance does not have pointers, recursion and record structures. It is with this background, that we attempt to find ways to represent OO features in Occam. It is not an attempt to change the language structure to accommodate OO features, rather it is an attempt to build OO structures on top of the existing language structure - a method of organising programs OO style. Such a method supports an object-oriented way of programming but does not enforce it.

6.4.1 Classes, objects and encapsulation

Our definition of an object is that of an entity that encapsulates both services as well as data. A class in this context, would be a static description of a typical object. In Occam, such a class/object definition closely resembles the Occam procedure PROC. There are however subtle differences between an Occam process and an object as defined. The similarities and differences between objects and processes have already been pointed out (chapter 3, section 3.2.4.2) and will be elaborated below.

Differences:

a) Objects are not inherently concurrent whereas processes are.

b) Processes do not include inheritance.

c) Encapsulation in objects serves to hide information and manage the complexity of a system whereas encapsulation in processes "express the ability of to execute programming modules independently." [Thomas 89].
d) Instantiation in objects is a dynamic activity whereas in processes - at least in OCCAM processes instances are statically defined by the PAR construct.

Similarities:

a) Processes are composed of code (syntactic expression) and data while objects consist of class and instance.

b) Objects use message passing while processes use communications.

With this in mind, we define an Occam class and object as follows:

A class is an Occam PROC with the following characteristics:

i) set of well defined services which are themselves processes,

ii) an interface consisting of a tagged protocol whose fields identify the class's services,

iii) a message selection mechanism and

iv) Private functions and data.

An object is a run-time instance of a class as defined above.
An illustration of it is shown (fig. 6.1)

Figure 6.1 *An illustration of the structure of a class in Occam.*

The Occam representation of a typical class is shown below.

```occam
PROC Classprocess (CHAN OF CLASSPROTOCOL classchan, outchan)
[] [] BYTE OF PrivateData:
PROC PrivateFunction(parG,parH)
:
PROC Service-1(parA,parB,outchan)

..............
..............
:
PROC Service-2(parX,parY,outchan)

..............
..............
:
```

89
ALT (the message selector)
classchan ? CASE
tag1;[x]BYTE;[y]BYTE
    Service-1(p1,p2,outchan)
tag2;INT::[]BYTE;INT;[z]BYTE
    Service-2(p3,p4,outchan)
:

Figure 6.2: The general structure of a class.

The Class defined above is an Occam procedure with the name Classprocess. Classprocess may have more than one channel parameters however the incoming channels must be of type CLASSPROTOCOL which is a variant protocol whose tags identify Classprocess's services.

The structure of CLASSPROTOCOL is shown below.

PROTOCOL CLASSPROTOCOL
    CASE
tag1;[x]BYTE;[y]BYTE
tag2;INT::[]BYTE;INT;[z]BYTE
:

More than one other classes may communicate with Classprocess but they must do so using a Classprocess protocol channel. A message received by Classprocess is trapped by the message selector which is defined in terms of the Occam ALT construct.
A message such as

```
tag2;30::[passenger FROM 0 FOR 30;[Flight FROM 0 FOR 3]
```

for example will cause the message selector to invoke Service-2. A similar message with tag1 will cause Service-1 to be invoked. Restricting incoming channels to be of type CLASSPROTOCOL and having a tightly controlled message selector enforces encapsulation. The private functions and data can only be accessed by services of the class.

### 6.4.2 Message Passing.

Message passing has already been touched on in the above example. More generally, we should stress the difference between message passing in traditional OO systems where there is just a single thread of control and message passing in parallel systems. The notion of message passing in sequential OO systems is an indexing mechanism to access object methods and serves to enforce encapsulation. In parallel systems, where there is more than one thread of control simultaneously active, it is more than that; it means sending messages through channels whether internal or physical, and involve routing, synchronization, acknowledgments, multiplexing and other such characteristics of message passing in communication networks.

The general format of a message in our model explicitly indicates the method to be invoked, and the parameters to be used. The object whose method is to be invoked is not indicated in the message because channels connecting objects are statically defined in Occam. The Occam variant protocol is used to express a message format as in the example CLASSPROTOCOL shown above. A class's protocol is its interface to the outside and
shows the services it can provide. Message calls must be of the right protocol format otherwise they are rejected.

6.4.3 **Instantiation, deletion and object managers.**

An instance of the class comes into existence when a class is invoked. For many classes there is just one instance, however one can have multiple instance classes in an application. The Occam PAR construct is used to express such multiple instance classes as shown in the following example.

```occam
VAL MAXOBJ IS 50:
PAR i = 0 FOR MAXOBJ
  Classprocess(classchan[i],.....)
```

In the above case MAXOBJ instances of *Classprocess* are declared and once executing, MAXOBJ instances will become active and start executing. Notice that the upper bound on the number of active instances is predetermined. This static feature of Occam is a constraint which has to coped with or some way found to get round it. One way to get round the problem is to create an object manager which would manage the number of instances of a class. This is the approach taken in this project; for every multiple object instance, there is an object manager. The maximum number of instances of a class is predetermined but the manager can manage the number of instances which are actually active at any one time by deleting or adding new instances so long as the total is not greater than the maximum.
An example of an object manager is shown below:

PROC Objmanager(CHAN OF OBJMAN obchan, outchan)
VAL MAXOBJ IS 100:
[MAXOBJ] INT NumInstances:
[MAXOBJ][10] BYTE Objlist:

PROC Delete(objname,outchan)
  If NumInstances is greater than 1
    delete one instance
    Reduce NumInstances by 1
  else
    return

PROC AddNewObj(objname,outchan)
  If NumInstances is less than MAXOBJ
    Add new instance
    Increase NumInstances by 1
  else
    return

-- Begin
ALT
  objchan ? CASE
    new;10::[objname FROM 0 FOR 10]
      AddnewObj(objname,outchan)
    del;10::[objname FROM 0 FOR 10]
      Delete(objname,outchan)
  num outchan ! NumInstances
: -- END of Objectmanager.

Note the tags new and del which indicate the invocation of Adnew and Delete methods respectively. The tag num is a message to return the number of currently active instances.
6.4.4 Inheritance.

The main reason for using inheritance in most OO development environments is for code sharing. This sharing behaviour is enforced by polymorphism and dynamic binding. Although most authors agree that it is not a necessary feature of OO systems, it is nevertheless a highly desirable one. There are one or two problems and contradictions about code sharing through inheritance that need to be pointed out here.

i) Code sharing between classes reduces the degree of encapsulation; this is against the spirit of the OO paradigm.

ii) With systems that support dynamic typing, such code sharing can lead to conflicts.

Suppose for instance that we have a subclass B that inherits a method X from superclass A and another class C which is a client of class B. Now if superclass A drops method X from its interface, then class B will also drop this method accordingly. However as far as class C is concerned class B still has method X in its interface.

The problem of code sharing through inheritance is more acute in a parallel environment where an inheritance hierarchy may be distributed over many processors. In such case, accessing an inherited service that resides on another processor can be difficult; it is not possible to use an indexing mechanism such as used in C++ (Section 2.2.1.3) because memory is not shared. Some alternative solutions that other people have tried to solve this problem have already been discussed (section 3.1, iii). In addition to these alternatives, we could have code migration. So if for instance a class needs to access an inherited service it would search the inheritance hierarchy across processors and when it finds the service, it physically transfers it. This alternative can lead to message overheads and reduced performance of the system.
It is interesting to note that others for instance [America 87] in POOL-T and [Annot and Haan] in POOL and DOOM have opted to leave out altogether the implementation of inheritance hierarchies giving reasons about the unclear semantics of inheritance. In our case, we attempt to incorporate inheritance into the application as a code sharing mechanism in two ways. These suggested methods are ways of enforcing inheritance which do not involve code migration and the associated problems of message overheads.

**Inheritance alternative 1.**

The first alternative is used in situations where the inherited service does not have to access the inheriting class's private variables. If a subclass B requires a service X from superclass A which has to access subclass B's private variables then this alternative is not used. It would typically be used in a situation where a message is passed to a superclass, some processing would be done (in the superclass) on the parameters passed, and the results would then be returned to the subclass.

**Inheritance alternative 2.**

This alternative would be used in situations where an inherited service has to access a subclass's private variables. In this case, we create a separate class which we call the image base of the inheritance hierarchy. This class would contain all the services of all classes in the inheritance hierarchy and their private data types. This image base would be placed in one processor while the member classes of the inheritance hierarchy may be distributed over many processors. A class in this hierarchy would contain only pointers to the image base or to its superclass.
An example of such an inheritance hierarchy is shown (fig. 6.3)

Suppose in the above example \textit{classX} requires a particular service \textit{SI} of \textit{subclass}1. It would then send a message to \textit{subclass}1 indicating the service required. If \textit{subclass}1 has to inherit \textit{SI}, it would pass on the message to \textit{superclass} which would then send the appropriate message to the image base to perform the service. On the other hand if \textit{SI} is one of \textit{subclass}1's own services, it would directly send a message to the image base to perform the service.
6.4.5 The effect of removing inheritance from the implementation.

As stated, the main reason for incorporating inheritance into the application is for code sharing. It should be born in mind however that the whole exercise of the implementation is to translate the conceptual designs derived in Fig. 5.9 and Fig. 5.10 into an application that reflects what is modelled in these designs. Thus the code sharing function is built in within the framework of the semantic relationships between classes. It is not code sharing between unrelated classes because it improves efficiency.

In a parallel architecture environment, it is often the case that classes which are related and therefore part of one inheritance hierarchy, are distributed across many processors. Obviously if there is no mechanism for sharing common code in such case then code will have to be replicated in many classes. But incorporating such a code sharing mechanism is however not that easy given the added complexities introduced by the parallel nature of the application. Introducing code sharing within the framework of the semantic relationships between classes must be analysed in the context of efficiency as well as conceptual organisation. Beyond a certain point the two can be said to be mutually exclusive. If we choose to ignore the semantic relationships between classes and code the application in such a way as to increase efficiency, we will sacrifice clarity and conceptual organisation. It is clear then that the cost of clarity and conceptual organisation needs to be balanced against efficiency.

The incorporation of inheritance in this application serves to
allow sharing of code as well as to enforce the semantic relationships between classes. The obvious problem that this introduces is that of message overheads which increase with the volume of transactions that take place per unit time and which in effect reduces efficiency. Of the two inheritance alternatives suggested in section 6.4.4, alternative 1 is the one that introduces less message overheads and contention problems. Alternative 2 may lead to contention problems for large inheritance hierarchies.

If inheritance is removed from the application, the application will not fall apart. There will clearly be less overheads in message passing and efficiency will increase. It however means that code has to be replicated for some of the classes. In this particular application, there are not that many methods to be shared and each method is small in terms of memory size. However one can imagine a hundred classes sharing five common methods and the amount of replication that need to be done if inheritance is not used. As the functionality of certain parts of the application change over time, modifications must be made and modifying replicated code can be tedious. This is especially so in an Occam environment, where modifications to functionality must necessarily accompany modifications to intra- and interprocess communications channels. The other problem that will occur if inheritance is not incorporated is that there will be no clear logical relationships between classes. The application will probably work more efficiently but only because it has been physically structured to do so.
6.4.6 The implications of using static objects rather than using dynamic instantiation.

We reiterate again that the whole exercise of the implementation is to translate the designs derived in Fig.5.9 and Fig.5.10 into an application that reflects what is modelled in these designs. The incorporation of dynamic instantiation is thus an attempt to model the instantiation behaviour of an object-oriented parallel system with multiple instance classes. It is clear that how dynamic instantiation is implemented depends on the implementation environment; in our case the Occam/transputer environment. We saw in section 6.4.3 one way in which dynamic instantiation could be implemented in Occam. The syntax of the PAR construct requires there to be a predefined upper limit on the number of possible instances. This means that memory is set aside for each instance whether or not it becomes active during the course of execution. Obviously an increase in the number of instances means more memory. The main advantage of using the PAR construct is that it allows for the creation and parallel execution of multiple instances. From a coding point of view, the PAR construct is a much more neat and elegant way of defining a multiple instance class compared to say Ada. It also makes channel communication much easier to define; a channel vector can be elegantly defined which would correspond to the number of instances of a class and thus be able to handle messages to or from multiple instance classes easily. An important feature of communication with multiple instance classes that has become apparent in the implementation exercise is knowledge of the
status of a multiple instance class. Object managers were created for this purpose - to maintain knowledge of the status a multiple instance class. In our case their function was limited to keeping track of the number of active instances however it could be further developed to include further information about individual instances.

If static objects are used instead of dynamic instantiation then obviously there would be no multiple instance classes as defined here. Instead each instance of a class would be hard coded. So for instance if in the system there are five actual travel agents, which conceptually belong to class Travelagent, then we would have five separate travel agent classes coded into the program. From an efficiency point of view, this restructuring would mean that object managers would not be needed and therefore message passing between classes would be faster. It would also mean the individual hard coded instances can be distributed to different processors to balance communication in the network. However we sacrifice conceptual organisation in this restructuring. Other disadvantages would mainly be in modification of the implementation. For instance, every time a new travel agent is created, a new class has to be hard coded into the program and the communication channels have to modified accordingly. This can be messy in Occam; if twenty travel agents need to communicate with the class Reservation then twenty uniquely defined channels have to be defined in Reservation to allow for this communication let alone the implications this will have for parallel communication within the network. An equally messy situation arises if we decide to
delete a hard coded instance of a class.

So there are both advantages and disadvantages in using static objects rather than dynamic instantiation. Once again it is apparent that like in inheritance, the cost of clarity and conceptual organisation of the implementation has to be balanced against efficiency.
This structuring of the inheritance might appear unnecessary but it does enforce inheritance. The services of subclass1 and its superclass(es) for instance are visible to classX through subclass1's interface and similarly for subclass2. Furthermore ClassX and classY could not careless where the services reside nor how they are accessed and performed. ClassZ on the other hand only has the services of superclass visible to it. Since all the services and data types reside on the same class and in one processor, there is no problem accessing the private data of individual classes and more importantly no code need migrate to other processors thus avoiding the problems associated with message overheads.

6.5 HOW THE DESIGN WAS IMPLEMENTED.

6.5.1 Special software and hardware considerations.

The design was implemented on the Occam2 tool set using a network of various configurations including 1 transputer, 3 transputers and 4 transputers of the IMS T800 series connected to an IBM PC. Implementation was done on the host computer, tested on the root transputer before being distributed over the network. The network was configured into a tree topology as shown in Appendix D.

6.5.2 Implementing classes.

The design shown in fig. 5.10 was followed very closely in the implementation. Several classes including Query, SidFares, amrouter and SpecFares were left out to simplify the implementation - without losing significant design features. Each class, multiplexor and router was implemented as a separate file. In this way, allocation of classes to processors and load balancing could be easily facilitated. Class protocols were also kept as separate files or
grouped where all the corresponding classes were closely related and likely to end up in the same processor.

### 6.5.3 Allocating classes.

Only 4 transputers were available during implementation, and this meant that the number of different possible configurations was limited. Figure 6.4 shows a configuration to be implemented on 4 transputers. It is similar to the design in fig. 5.13 except that the cluster *Query* was moved into the class *ARSystem* so that it is in the same class as the other classes that communicate with the host, i.e., *Reservn, Flight* and *Fares*. *ARSystem* is the only classes according to this configuration that communicates with the host.
Figure 6.4 *Partitioning of design to be implemented in 4 transputers.*
Figure 6.4 shows the actual configuration of the design into 4 transputer. The labelled arrows are the names of the physical links.

Figure 6.5 *The design configured to run in 4 processors.*

We also tried to configure the system to run on one, two and three transputers. An outline of the configuration description for 4 transputers are shown below:

```plaintext
-- {include files
--}
-- {inter-node channels
--}
-- {configuration description

PLACED PAR

PROCESSOR 0 T800

    -- {EXTERNAL CHANNELS
PLACE  links
    --}

SEQ

   Arsystem(fs,ts,aquery,tquery,aresp,tresp)
```
The PLACED PAR is the Occam construct for distributing classes to processors. The processors numbered PROCESSOR 0 to PROCESSOR 4 refer to transputer nodes on which the classes will physically be after distributed. As can be seen above, PROCESSOR 0 will run Subsystem ARSystem. Similarly PROCESSOR 1 runs subsystem Tragent, PROCESSOR 2 runs Amanager and PROCESSOR 3 runs Gparser. Refer to fig. 6.4 and 6.5 to confirm the partition and configuration. Details of other configuration descriptions are in appendix C.
6.5.4 **How does the system work?**

The resulting object-oriented parallel system after implementation, consists of a collection of classes and their support objects - multiplexors and routers, which are physically distributed throughout the network according to a configuration as for instance in fig. 6.5. All objects execute in parallel and become active immediately the system is started. Objects communicate by passing messages through channels and to avoid deadlocks - a common problem in parallel systems, each object has been required to explicitly wait for an acknowledgement to every message sent out before proceeding with other tasks.

Although it is a simplistic model of a real airline reservation system, this model does emulate the behaviour of such a system and serves to demonstrate the ideas put forward about object-oriented parallel systems.

In a typical working session, the system would operate as follows: Travel agents and airline managers would be making queries in parallel and continuously. Each query would first be edited and then parsed to check its format against standard query formats. Once a query is parsed and verified, it is forwarded to the airline reservation systems interface which identifies what type of query it is: whether a reservation, a cancellation or the addition of a new flight, and then sends the query to the appropriate object that provides that service. Upon completion of a service or in the case where there is an error, some message might be displayed to that effect and an acknowledgement is then sent back to the user who originated the query. Since objects are executing in parallel, there would be many threads of control running in parallel.
6.5.4.1 A single thread of execution.

Let us trace a single thread of execution to see what takes place - it would be helpful to refer to fig. 5.10 in chapter 5 to graphically trace the execution (note however that because of restrictions imposed by Occam during the allocation process, \textit{Tr\_objman} communicates with \textit{Qtrmux} instead of \textit{Trquery} as shown).

Suppose that a query is made by a travel agent to make a reservation. The following sequence of operations would take place.

i) The query would originate from the class \textit{Trquery}. A typical reservation query at this stage would be a string of the form

\begin{verbatim}
 to.tragent ! rf;40::[Buffer FROM 0 FOR 40];query-num
\end{verbatim}

where \textit{to.tragent} is the channel through which the query is to be sent, \textit{rf} is a tag indicating it is a reservation query and \textit{query-num} is the query number. This query would then be multiplexed (because there are many instances of \textit{Tragent}) at \textit{Qtrmux} and then sent to an instance of \textit{Tragent} say TRAGENT05. During multiplexing, the number of active instances of \textit{Tragent} would be checked at \textit{Tr\_objman} the object manager. \textit{Tr\_objman} keeps track of the identities of instances of \textit{Tragent} and the number that are currently active. ii) When the query gets through to TRAGENT05, it is edited and then demultiplexed at \textit{Trpmux} and sent on to \textit{Trparse}. Editing results in splitting the query to the right format for parsing. The query after editing would look like

\begin{verbatim}
 rf;flight.no;agent;date:size::passenger.
\end{verbatim}

This is in the format of a \textit{Trparse} channel protocol.
iii) At \textit{Trparse}, the query is split into its components and parsed separately. There are different services to parse each component of the query. In this case, the service to parse passenger name, \textit{is.passenger}, is local to \textit{Trparse} but services for parsing \textit{flight.no} and \textit{date} are inherited from the superclass \textit{Gparser} and have to be accessed. Since parsing consists of just checking the correct syntax of \textit{flight.no} and \textit{date}, \textit{inheritance alternative 1} (section 6.4.4) is used. The outline of \textit{Trparse} class is shown here.

\begin{verbatim}
PROC Trparse(CHAN OF SP fs, ts, CHAN OF TRAPRO from.tragent, CHAN OF ACK to.tragent, CHAN OF PARSEP to.parse, from.parse,
CHAN OF TRAPRO to.trout, CHAN OF ACK from.trout)

PROC is.passenger([]BYTE name, INT ln, BOOL resp) (local method)

Parse name :
SEQ
WHILE TRUE
SEQ
ALT (message selector)
ALT (message selector)

from.tragent ? CASE
rf; flight.no;agent;date:size::passenger (incoming query)

--{} parse reserve query
SEQ

(Methods to parse \textit{flight.no} and \textit{date} have to be inherited from superclass \textit{Parser})
to.parse ! pr.fl;flight.no
to.parse ! pr.dat;6::date
get response from superclass

(method to parse passenger is local)
is.passenger(passenger, size, ok2)

IF
All parts of the query are ok then
pass verified query on
wait for ack
propagate ack back to user.
TRUE
Query is invalid
propagate ack back to user.

Deal similarly with other queries

--{}))
\end{verbatim}
iv) After query is parsed and verified, it is sent to the Tr_Interface where the type of the query is identified (in this case it is a reservation query) and then sent to Reservn. Before a reservation is made, the flight number and the number of vacant seats are checked. This means communicating with class Flight. A message to check the flight number and number of vacant seats is thus sent to the flight hierarchy through FlightRouter. The message finds its way up the Flight inheritance hierarchy where it ends up in Flimage - the image base of the Flight inheritance hierarchy. The appropriate services (Findflight and Checkbking) are then invoked and the results sent back to Reservn. If the flight exists and there are vacant seats, then the reservation is made, a message confirming the reservation is displayed and an acknowledgement to this effect is sent back to the originator of the query - TRAGENT05. Once the acknowledgement is received by TRAGENT05, then it can proceed with other tasks - thus ending this thread of execution of the reservation query. During the execution of the system, many such threads of control will be running in parallel. Outlines of the implementation of some of the classes in the model are shown in addendix C.

It is interesting to see how the flight inheritance hierarchy is implemented (refer to fig. 5.10 for the design). The implementation of the flight inheritance hierarchy is an example of Inheritance alternative 2 (section 6.4.4) because each of subclasses Mflight, and Oflight have their own private data and inherited services may have to access their private data. A message to Mflight to find a flight for instance would cause it to inherit method Findflight from superclass Flight. Findflight would then have to search through Mflight's flight list (a private data).

The skeleton structures of Mflight shown below.
PROC Mflight(CHAN OF SP fs,ts,CHAN OF FLIGHTPRO from.mux,CHAN OF ACK to.mux,CHAN OF FLIGHTPRO to.fliimage,to.flight,CHAN OF ACK from.fliimage,from.flight)

SEQ
  WHILE TRUE
  ALT
    from.mux ? CASE
      add.f1:f1.no2;cap1:book1;numlegs:who2.id
      SEQ
        to.flight ! add.f1:f1.no2;cap1:book1;numlegs:1
        wait for ack
        send ack back to caller
      SEQ
      add.leg:f1.no5;cost:who5.id
      SEQ
        to.fliimage ! add.leg:f1.no5;cost:0
        wait for ack
        send ack back to caller
      SEQ
      chk.bking:f1.no7;who7.id
      SEQ
        to.flight ! chk.bking:f1.no7;who7.id
        wait for ack
        send ack back to caller
    Deal similarly with other queries.

Notice that the method to add a flight has to be inherited from the superclass Flight, the message is propagated upwards whereas a message to add a leg to a flight schedule is sent directly to Flimage, the image base of the flight hierarchy. Mflight like all the other classes in the flight inheritance hierarchy does not do any processing. It just filters messages and either sends them up the hierarchy if the required services are inherited or sends it directly to Flimage.

The protocol for Mflight as indicated by parameter highlighted is FLIGHTPRO and is shared by all the classes in the flight inheritance hierarchy - Oflight, Flight, Mflimage. Its structure is shown below.
PROTOCOL FLIGHTPRO

CASE

find.fl:[3]BYTE;INT -- flight.no, caller.id
-- fl.no, capacity, booked, nlegs, caller.id
drop.fl:[3]BYTE;INT -- fl.no, caller.id
disp.fl:[3]BYTE;INT -- fl.no, caller.id
add.leg:[3]BYTE;INT:INT -- fl.no, cost, caller.id
drop.leg:[3]BYTE;INT:INT -- fl.no, cost, caller.id
chk.bking:[3]BYTE;INT -- fl.no, caller.id
updat.bking:[3]BYTE;INT -- fl.no, caller.id

The rest of the protocols of other classes is given in appendix C.
CHAPTER

7

Testing and evaluation.

7.1 INTRODUCTION.

In this chapter, an attempt is made to assess the extend to which the aim of the project as set out in chapter 4 has been achieved. To a large extend, our evaluation would be more of a qualitative nature than quantitative because it is an attempt to take a critical look at the main issues - advantages and constraints, identified in the translation from design to implementation (Chapter 5 and 6) of the OOD.

7.2 QUANTITATIVE ANALYSIS.

[Levitan 87] gives some good metrics for measuring the efficiency of Parallel Architectures and algorithms. These are however best applicable to large networks of processors. Our case is a trivial one of 4 processors. In any case our emphasis is not on parallel speedup but rather on how it compares with similar OO and parallel systems. We give below some quantitative assessment of the project.
7.2.1 How much code and how long did it take? 
Recall from chapter 6 that all classes were coded into separate files to ease allocation in the distribution process. Taken together, the total number of lines of source code is in excess of 2000 lines. This took about 250 manhours to code. Part of the reason for this rather long period is attributed to the poor software development support. In particular the debugger for the Occam tool set was not helpful, and debugging parallel programs without a good debugger is not fun!

7.2.2 Performance
Measuring the performance of the Object-oriented parallel system implemented in chapter 6 implies measuring its performance in comparison with typical object-oriented systems and parallel systems. The object-orientedness of an application is a difficult thing to measure in a quantitative manner; this is more of a qualitative assessment and so will be looked at in the next section. With Parallel systems, the obvious measurement is for parallel speed up. We could also look at typical parallel systems characteristic behaviour such as deadlock prevention, proper termination and non-determinacy.

7.2.2.1 Timing Results.
As mentioned above, it is important to get some assessment on the performance of the implementation. In particular we want to show that the translation process results in a system with the advantages of OO systems and parallel systems and whose performance is comparable with current OO or parallel systems.

One way to assess the performance of such a system is to time the interval between send and receive messages of an object. With the way the system is implemented, we time the interval that each instance of a Travel Agent and Airline Manager send and
receive the answer to a query. The timings were first done on the host system and then on the network with different configurations. Details of the tests are shown in appendix E. A summary of the results for some of the queries are shown below (fig. 7.1)

Test results

7.1 a) Managers Queries with host communication included.

<table>
<thead>
<tr>
<th>Qr No</th>
<th>Host</th>
<th>Root-T</th>
<th>3-Transputers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1991</td>
<td>1979</td>
<td>1972</td>
</tr>
<tr>
<td>1</td>
<td>279</td>
<td>270</td>
<td>262</td>
</tr>
<tr>
<td>2</td>
<td>308</td>
<td>296</td>
<td>286</td>
</tr>
<tr>
<td>3</td>
<td>311</td>
<td>299</td>
<td>289</td>
</tr>
<tr>
<td>4</td>
<td>310</td>
<td>298</td>
<td>288</td>
</tr>
<tr>
<td>5</td>
<td>315</td>
<td>304</td>
<td>293</td>
</tr>
<tr>
<td>6</td>
<td>289</td>
<td>279</td>
<td>268</td>
</tr>
<tr>
<td>7</td>
<td>281</td>
<td>270</td>
<td>261</td>
</tr>
<tr>
<td>8</td>
<td>280</td>
<td>268</td>
<td>260</td>
</tr>
<tr>
<td>9</td>
<td>147</td>
<td>140</td>
<td>135</td>
</tr>
<tr>
<td>10</td>
<td>151</td>
<td>144</td>
<td>138</td>
</tr>
</tbody>
</table>

7.1 b) Managers Queries without host communication.

<table>
<thead>
<tr>
<th>Qr No</th>
<th>Host</th>
<th>Root-T</th>
<th>3-Transputers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1759</td>
<td>1734</td>
<td>1731</td>
</tr>
<tr>
<td>1</td>
<td>17</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>17</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>17</td>
<td>17</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>18</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>18</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>18</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>18</td>
<td>18</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>19</td>
<td>19</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>20</td>
<td>20</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>21</td>
<td>19</td>
<td>17</td>
</tr>
</tbody>
</table>

*Figure 7.1 Timing results for manager queries.*

Timings are in terms of the number of ticks which can easily be converted to seconds if desired.
7.2.2.1 Timing Results (further explanations to results).

The timing results in Fig. 7.1 have perhaps not been explained clearly. Communication with the host as referred to in this section refers to communication with the host for the purpose of outputting information; in this case onto the screen. The system must communicate with the host for input information. Input information in this case is stored in a file in the format as shown in appendix E1 pg 153, and is read from the file by the host as and when required. The comparison made is between the system as it operates without outputting information onto the screen and outputting information onto the screen. An example of output information is shown in Appendix E2 pg. 153. The first column of fig. 7.1 a) labelled Host, shows readings in the case where the host executes the application and outputs information to the screen. The corresponding first column in fig. 7.1 b) shows readings in the case where the host does not output information onto the screen. Similar comparisons are made with the system configured to run on the root transputer, three transputers and on four transputers. The way the application is structured does not allow it to run on two transputers without complete restructuring.

The main reason for the timings and comparisons is basically to find out how the object-oriented parallel application, which is a translation of the designs in fig. 5.9 and 5.10, would perform under different configurations of the transputer network. As regards efficiency, it would have been ideal to code a non-object oriented parallel version and an object-oriented non-parallel
version of the application to compare with this one. This in itself is I think another project.
These manager query timings were done when queries were being executed in parallel with other query types and not in isolation. As can be seen, there is a consistent trend of performance increase albeit a very small one, as the system is distributed into more processors both in the case where the system communicates with the host (fig. 7.1a) and without the host (fig. 7.1b). More details of the tests are shown in Appendix E. The point here is that there has not been a degradation in performance as a result of the design translation.

7.2.2.2 Non-determinacy.

A major characteristic of parallel systems is non-determinacy - the fact that processes can execute in parallel. In our model, all objects run in parallel and immediately become active when the system is started. As such there are many threads of control executing at once thus exhibiting a parallel nature.

7.2.2.3 Deadlocks and termination.

Deadlocks are a potential problem in parallel systems. In our model, this has been prevented by making it a requirement for every object to wait for a response to every message sent out. Such requirement sometimes slows down the system a bit especially if a certain amount of processing is done before a response is sent back to a caller. However it gives the programmer more control over the execution of the application.

Termination is a more tricky problem to deal with because it has to begin at some point, propagated upwards in an object or process and then propagated to all other objects or processes concerned. This requires objects to have a certain amount of
global knowledge as for instance the number of active objects etc. It has thus not been dealt with in this implementation.

7.3 QUALITATIVE ANALYSIS

7.3.1 A critical analysis of the method.

In this section we try to analyse the OOD method proposed in this project and its subsequent implementation.

7.3.1.1 The OOD method.

The obvious difference between our OOD method and those of others discussed is that our method tries to capture parallel features as well as OO features of an application. It is based on other methods (see chapt. 5) but certain features have been emphasized in order to bring out the parallelism in an application. For instance our definition of a class as encapsulating services as well as data predetermines the unit of parallelism for allocation to processors. Modelling objects as defined in Smalltalk would imply a different unit of parallelism. Another set of features which are emphasised for parallelism are the relationships between entities. Relationships imply object interaction and in a parallel environment, the nature of these relationships are important. Two particular relationship that appear to be important in the allocation process are the generic and aggregate relationships because they provide a natural grouping for clustering related classes.

The notations used in the design process are we believe rich enough to express OO features and parallel features of an application.

Most OOD methods begin by identifying classes and then move on to define services and and the rest. Our method involves a blend of structured analysis modelling (DFD) and the ER model which some might consider an unlikely
combination. We have found it useful to use structured analysis to decompose active entities because entities do encapsulate behaviour and the best way to model this is by using DFDs. Decomposition assists in identifying the services of a class. It forces the designer to focus on the nature of the class's behaviour by identifying the public services and private methods, by identifying the classes private data and by identifying the other classes with which this class interacts. Apart from this, decomposition also assists in identifying other entities. It is often the case that the initial ER diagram contains entities which are the obvious once in the system. In the process of decomposing the initial active entities, other entities may be discovered. In most cases these other entities are probably closely related to the initial entity. Such close relations help in deciding how to cluster classes to allocate to processors later on. One difficulty with decomposing to find entities is how to tell if you have found one. It is very similar to the problem of trying to construct an conceptual ERA schema from a DFD - the problem of getting a two dimensional representation of a three dimensional DFD. This calls for familiarity with the problem domain but clues can be got by looking at clusters of data flows around data stores. This is the method used here. Obviously with bigger problems data stores and data flows need to be looked at more closely.

The first three stages of our design are independent of target implementation domain and so may generally applied to any problem that contains parallel as well as OO features. The last two stages involve mapping the design into a domain specific logical design. Now there are many possible implementation domains but there are also applications which are not suitable for some implementation domains. We can thus get the following possibilities (fig. 7.2)
Problem 1 from the diagram above may only be suitable only for Ada, similarly problem 2 for the transputer/Occam and problem 3 for the POOL and DOOM environment. We discuss more about the implementation environment below.

7.3.1.2 The transputer/Occam implementation of the OOD.

There are certain constraints imposed on the implementation of the design by the transputer/Occam environment, which need to be emphasised. The main hardware constraints have been those associated with point-to-point communication and the limitation the number of physical links per transputer.
node. Point-to-point communication means that some sort of switching mechanism must be employed to route message to nodes not directly adjacent. This problem was not difficult in our model because of the small number of nodes involved, however it is one which needs to be considered in large systems. The limit in the number of physical links means that if there are more than 4 processes that need to perform inter-process communication, then some multiplexing must be done.

Occam performs well as a language for parallel processing, however it imposes certain constraints with regard to the implementation of OO features. Basically Occam is a static language. It for instance supports only static bounds on arrays and FOR loops and inter-object channels are also defined statically. Also because of its design philosophy of 'simplicity', Occam lacks many of the important data types that other languages such as Ada have. It for instance does not have recursion, record data types, user defined types, pointers etc. These limitations have meant that ways have to be found to represent OO features Occam. A good example of this is the construction of an object manager to manage instances of a multiple instance class because processes are unnamed. Although it does solves the problem it is clearly an expensive solution.

It is clear from the preceding discussions that the main problems encountered in the Transputer/Environment have been those of trying to accommodate the OO paradigm in this specific distributed memory architecture environment.
7.3.2 Other Implementation domains.

We note in fig. 7.2 that there can be many possible implementation domains.

**POOL and DOOM**

The POOL and DOOM implementation domain is an interesting comparison to the transputer/Occam because here we have a hardware DOOM which is built especially to support POOL programs. An interesting feature of DOOM is that it has a separate communication processor layer from the main processor layer (see chapter 3, section 3.2.3.2). This layer takes care of the message passing network quite independently of the data processor layer. It is a solution of the point-to-point problem encountered in the transputer. POOL of course is a language designed especially to incorporate OO features as well as parallel features of an application. We therefore do not have the problems that arise as in Occam. The main problem is that such systems are quite expensive and not really general purpose.

**ADA**

Ada [Booch 83], is probably another interesting implementation domain. Ada is not an object-oriented language (it is object-based) however, it supports both the object-oriented paradigm and concurrent programming. It has a lot more features than Occam [Burns 88] including scalar types, multidimensional arrays and records. It also supports dynamic creation and deletion of objects. Processes in Ada communicate using tasks which are not parameterised as Occam PROCs are. Ada processes are named unlike in Occam. Despite these advantages, there are several drawbacks in using Ada as an implementation domain. Firstly Ada is not design for mass parallelism; it does not have the support of an environment like the transputer with point-to-point links that provide the possibility for parallel scale-up. Secondly Ada
processes may communicate via shared variables which is against the spirit of Object-oriented parallel systems. Representing parallel processes is tedious task of defining tasks.

7.3.3 Other problem domains.
The problem domain chosen for this project is one which involves a lot of message passing; quite similar to other systems such as a Library system, Business data processing applications and distribute database systems. This is probably a problem domain which is quite difficult to implement in a transputer/Occam environment (at present) because of the limitation in the number of physical links between nodes and the point-to-point message passing system used. This is the main reason that there has not been a considerable speed-up after object allocation. The transputer/Occam environment has mainly been used for problems in which there is more processing than message passing. Examples of such applications include image processing, Artificial intelligence, Natural language translation, simulation and signal processing[Jamieson et al 87]. However modifications can be made and indeed have been attempted to the environment to accommodate intensive message passing applications.

Our design method is quite independent of the type of problem and so may be used to express features of any of the problem domains mentioned above.

7.4 Development environment.

It is very important to have a good development environment for any parallel application let alone an object-oriented parallel application. The Occam tool set generally has a god set of library facilities. However the debugging facilities have not been very helpful. The debugger is difficult to follow and even the simulator ISIM
proved difficult to use. This unfortunate problems have caused us many hours of tedious debugging.
Conclusions and Future.

An attempt has been made in this project to investigate the extent to which the transputer/Occam environment could be exploited to support object-oriented ideas. It is a specific investigation of the more general problem of finding ways of exploiting distributed memory architectures to support Object-oriented ideas. It was clear from the outset that to incorporate the OO paradigm into parallel systems, it was essential to identify both essential OO features as well as parallel features of an application. Such features have been identified (section 5.1 of chapter 5).

8.1 THE DESIGN

To capture these features, a design method was developed. The design method is based on some of the emerging object-oriented design that have been proposed by Bailin, Wirfs-Brock, Henderson-Sellers, Coad and Yourdon and Meyer but with extensions added to express the parallelism in an application. The design consists of five phases, the first three of which are independent of implementation domain. The last two stages consist of mapping the design to the implementation environment and then subsequently allocating objects to
processors. Two important features of parallelism that are captured in the design are the class - which identifies the unit of parallelism to be distributed and object relationships. The types of relationships in particular were important in expressing object interaction and clustering - two features that are important in parallel systems. For instance 1:N/N:1 and M:N relationships identified inter-object relationships that require multiplexing or demultiplexing in the Transputer/Occam environment and generic relationships identified natural groups of classes that can be used as a basis for clustering classes into subsystems for allocation to processors.

8.2 The implementation domain.

The transputer/Occam environment was chosen to implement the design primarily because of availability. However it is a suitable choice because in general, both the transputer and Occam are cheap and widely available. Also the transputer/Occam environment is one that is becoming more promising for developments in this direction especially with the rapid advancement of technology. Unlike POOL-T or POOL and DOOM, the transputer/Occam environment is not one which is specifically tailored for developing object-oriented parallel systems. The transputer is a distributed memory architecture designed in close association with the non-object-oriented parallel language Occam. Constructing object-oriented applications under such environments amount to representing OO features in the Occam language to be executed in a transputer network. Apart from the fact that Occam is a non-object-oriented language, it is also a static language and lacked many of the features that other languages had. Therefore ways had to be found to get around some of the constraints in order to represent the concepts of objects, classes, encapsulation, message passing and inheritance in Occam. It became clear at this stage of development that certain additional support structures such as multiplexors, demultiplexors and routers had to be incorporated into the implementation. These features are peculiar to the transputer/Occam environment but they do
bring out some of the general constraints imposed on the development of such a model by the implementation domain. Implementation basically was a matter of constructing an object-oriented parallel program on top of the existing structure of the language by making use of the parallel features of Occam and the OO structures defined. Such a method of implementation is supported but obviously cannot be enforced without making changes to the Occam language structure.

The resulting object-oriented parallel system model consists of a collection of classes and support objects - multiplexors, demultiplexors, routers and object managers which are clustered into subsystems and distributed in the network according to various topologies. Once started, the objects become active and begin executing immediately. Some timing results of a typical session of execution have shown that the performance is not degraded as a result of this method of constructing object-oriented parallel applications. In fact there was a slight performance increase.

This project has shown that part of our design method can be applied generally to any problem that has OO as well as parallel features. It has also shown that the transformation from design to implementation is dependent on the implementation domain.

8.3 Future Developments.

Object-oriented designs have emerged in response to a need to better express OO features of applications. With the increasing interest in developing object-oriented parallel systems, there will inevitably be an interest in developing design methods to better express features of such systems.
While there are constraints in the transputer/Occam implementation domain there is reason to be optimistic about further research in this direction because of the rapid changes that are taking place in transputers and Occam. The increase in the number of inter-node links in a transputer and the inclusion of record types into Occam for instance have been predicted to appear in future versions. This will certainly make the transputer more powerful and make Occam a more expressive language.
8.3.1 Execution Overheads.

The translation from design to implementation has created a certain amount of execution overheads such as for instance in the incorporation of inheritance and dynamic instantiation. There needs to be further experimentation to determine the extent, in terms of execution overheads, to which efficiency has been sacrificed in order to obtain better conceptual organisation of the application. For example, bearing in mind that conceptual organisation of the application has to be balanced against efficiency, does the introduction of inheritance, dynamic instantiation or criteria of process distribution justify structuring the application in this manner?
References.


APPENDIX A: DATAFLOW DIAGRAMS.

In this part of the appendix, the services of some of the entities identified after the decomposition process (Chapter section 5.4.3) are shown.

1) The Extended Information Flow diagram.

2. The first level decomposition of entity ManagerQuery.
3. The first level decomposition of entity *AirlineManager*.

![Diagram of AirlineManager decomposition]

4. The first level decomposition of entity *ManagerQueryFormat*

![Diagram of ManagerQueryFormat decomposition]
5) The first level decomposition of entity ARSystem.

6) First level decomposition of entity Fares.
7) The first level decomposition of entity *Cancel*

---

130
8) First level decomposition of entity *Flight.*
Definitions for some of the classes identified in the design process (see section 5.6). Some changes have been made to these during implementation but the general identities of classes remain the same.

<table>
<thead>
<tr>
<th>Class</th>
<th>Subclasses</th>
<th>Clients</th>
<th>Servers</th>
<th>Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trquery</td>
<td>none</td>
<td>tragent</td>
<td>Tr_objman</td>
<td>getquery</td>
</tr>
<tr>
<td>Superclass</td>
<td>Query</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>Subclasses</th>
<th>Clients</th>
<th>Servers</th>
<th>Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tragent</td>
<td>none</td>
<td>Tr_objman</td>
<td>Trparse</td>
<td>InterpretQuery, EditQuery</td>
</tr>
<tr>
<td>Superclass</td>
<td>none</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>Subclasses</th>
<th>Clients</th>
<th>Servers</th>
<th>Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tr_objman</td>
<td>none</td>
<td>Trquery, Tragent</td>
<td>Tragent</td>
<td>Delete, New, NumberAgents</td>
</tr>
<tr>
<td>Superclass</td>
<td>none</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>Subclasses</th>
<th>Clients</th>
<th>Servers</th>
<th>Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trparse</td>
<td>none</td>
<td>Tr_interface</td>
<td>Tragent</td>
<td>is-passenger, is-agent, is-flight, is-date</td>
</tr>
<tr>
<td>Superclass</td>
<td>parser</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>Subclasses</th>
<th>Clients</th>
<th>Servers</th>
<th>Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reservation</td>
<td>none</td>
<td>Tr_interface, Am_interface</td>
<td>Flight, Fares</td>
<td>Reserve, Cancel, AgentReservations</td>
</tr>
<tr>
<td>Superclass</td>
<td>none</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>Subclasses</th>
<th>Clients</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fares</td>
<td>Stdfares, Specfares</td>
<td>Stdfares, Specfares</td>
<td></td>
</tr>
<tr>
<td>Superclass</td>
<td>none</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

132
<table>
<thead>
<tr>
<th>Class</th>
<th>StdFares</th>
<th>Superclass: Fares</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subclasses</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>Clients</td>
<td>Tr_interface, Am_interface</td>
<td></td>
</tr>
<tr>
<td>Servers</td>
<td>mflight, oflight</td>
<td></td>
</tr>
<tr>
<td>Services</td>
<td>CalcStandardfares, Economy, BusinessClass, FirstClass</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>SpecFares</th>
<th>Superclass: Fares</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subclasses</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>Clients</td>
<td>Tr_interface, Am_interface</td>
<td></td>
</tr>
<tr>
<td>Servers</td>
<td>mflight, oflight</td>
<td></td>
</tr>
<tr>
<td>Services</td>
<td>Studentfares, Charteredflights, childfares.</td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX C: Examples of Classes, multiplexors, protocols, subsystems and Configuration descriptions.

C1. CLASSES

Outlines of some of the classes of the Object-oriented parallel system are shown below.

a) Class Tragent

PROC TravelAgent(CHAN OF TRAPRO from.man,
    CHAN OF ACK to.man,CHAN OF RAWDATA from.trquery,
    CHAN OF TRAPRO to.trquery,to.trparse,
    CHAN OF ACK from.trparse,CHAN OF NAGENT to.procman,
    CHAN OF ACK from.procman)

  going := TRUE
  WHILE going
  ALT
  from.trquery ? CASE
    dat:len::buf;agent
    SEQ
      IF
        compare.strings("ff",[buf FROM 0 FOR 2]) = 0
        SEQ
        --{{ ff
          Edit findflight query
          to.trparse ! ff;fl.id;agent
        ALT
        from.trparse ? CASE
          single.ack;ok;agent
          to.trquery ! ff;fl.id;agent
        --}}}
        compare.strings("rf",[buf FROM 0 FOR 2]) = 0
        SEQ
        --{{ rf
          Edit reservation query
          to.trparse ! rf;fl.id;agent;date;
          24::[buf FROM 16 FOR 24]
        ALT
        from.trparse ? CASE
          single.ack;ok:id
          to.trquery ! rf;fl.id;agent;date;
          24::[buf FROM 16 FOR 24]
        --}}}
    Deal similarly with other queries
b) Class Parser.

PROC Parse.item([]CHAN OF PARSEP item.in,item.out)

  ///<parse flight
PROC is.flight([]BYTE fl, BOOL resp)

  ///</parse flight

PROC is.integer([]BYTE date,INT ln, BOOL resp)

WHILE TRUE (Activate object)

ALT i = 0 FOR 2
  item.in[i] ? CASE
    pr.fl;flight
    SEQ
      is.flight(flight,ok)
      IF
        ok = TRUE
        item.out[i] ! isit.ok;TRUE
        TRUE
        item.out[i] ! isit.ok;FALSE
    pr.dat:len::date
    SEQ
      is.integer(date,len,ok1)
      IF
        ok1 = TRUE
        item.out[i] ! isit.ok;TRUE
        TRUE
        item.out[i] ! isit.ok;TRUE

c) Image base for flight inheritance hierarchy - Flimage

PROC Flight.image(CHAN OF SP fs,ts,CHAN OF FAREPRO to.fare,
  CHAN OF ACK from.fare,[]CHAN OF FLIGHTPRO,
  from.flclass,[]CHAN OF ACK to.flclass,
  CHAN OF INITPRO init)

  ///<Find flight method
PROC Findflight([]BYTE flight.no,[]BYTE Fsched,INT Nfl,
  CHAN OF FLIGHTPRO ffmsg)

  ///</Find flight method
PROC Addflight(CHAN OF SP f,t,[]BYTE fl.no,[]BYTE cap,INT bk,
   []BYTE nlegs,INT pid,CHAN OF ACK msg)

PROC Dropflight([]BYTE fl.no,[]BYTE Fsched,[]BYTE Fdet,INT Nfl,
   []BYTE ag,CHAN OF FLIGHTPRO dmsg)

PROC Change.schedule([]BYTE fl.no,INT cst,chng,pid,
   CHAN OF FAREPRO to.f,CHAN OF ACK from.f,to.class)

PROC DisplayFlight(CHAN OF SP f,t,[]BYTE fl.no,CHAN OF ACK msg,
   INT pid)

PROC CheckBookings(CHAN OF SP f,t,[]BYTE fl.no,INT pid,
   CHAN OF ACK to.class)

PROC UpdateBookings([]BYTE fl.no,[]BYTE Fsched,[]BYTE Fdet,INT Nfl,
   CHAN OF FLIGHTPRO msg)

SEQ
   WHILE TRUE (Activate object)
   PAR
      ALT i = 0 FOR 2
      from.flclass[i] ? CASE
         find.fl;flight.nol;pidl
         SEQ
         FIND flight
         IF
            (pidl = 0) OR (pidl = 2)
            Findflight(flight.nol,Flight.sched,Nflights,msg)
            pidl = 1
            Findflight(flight.nol,MFlight.sched,MNflights,msg)
         ALT
            msg ? CASE
f1.found
SEQ
to.flclass[i] ! single.ack;TRUE;pid1
f1.notfound
SEQ
to.flclass[i] ! single.ack;FALSE;pid1

add.f1:flight.no2;capacity1;booked1;numlegs;pid2
---{( add flight
SEQ
Addflight(fs,ts,flight.no2,capacity1,booked1,numlegs,
    pid2,to.flclass[i])
---})
drop.f1:flight.no3;pid3
---{( drop flight
SEQ
IF
   (pid2 = 0) OR (pid2 = 2)
   Dropflight(flight.no3,Flight.sched,Flight.det,
         Nflights,agent3,dmsg)
   pid = 1
   Dropflight(flight.no3,MFlight.sched,MFlight.det,
        MNflights,agent3,dmsg)
ALT
dmsg ? CASE
   fl.dropped
   SEQ
     to.flclass[i] ! single.ack;TRUE;pid3
   fl.notfound
   SEQ
     to.flclass[i] ! single.ack;FALSE;pid2
---})
add.leg:flight.no4;cost;pid4
---{( add a leg to flightschedule
SEQ
   change := 1
   Change.schedule(flight.no4,cost,change,pid4,to.fare,
          from.fare,to.flclass[i])
---})
drop.leg:flight.no5;cost1;pid5
---{( drop a leg from the schedule
SEQ
   change := 2
   Change.schedule(flight.no5,cost,change,pid5,to.fare,
          from.fare,to.flclass[i])
---})
disp.fl:flight.no6;pid6
DisplayFlight(fs,ts,flight.no6,to.flclass[i],pid6)
chk.bking:flight.no7;pid7
---{( Check bookings
SEQ
   CheckBookings(fs,ts,flight.no7,pid7,to.flclass[i])
---})
updat.bking:flight.no8;pid8
---{( Update bookings
SEQ
   ...
---}
IF (pid8 = 0) OR (pid8 = 2)
    UpdateBookings(flight.no8, Flight.sched, Flight.det, Nflights, msg4)

pid7 = 1
    UpdateBookings(flight.no8, MFlight.sched, MFlight.det, MNflights, msg4)

ALT
    msg4 ? CASE
        bk.updated
            to.flclss[i] ! single.ack; TRUE; pid8
        fl.full
            to.flclss[i] ! single.ack; FALSE; pid8
        fl.notfound
            to.flclss[i] ! single.ack; FALSE; pid8
    end

: 

d) Class Reserve.

PROC Reservation(CHAN OF SP fs, ts, CHAN OF FLIGHTPRO to.flight,
    CHAN OF ACK from.flight, to.trouter,
    CHAN OF RESERVPRO from.trouter, CHAN OF INITPRO init)

    {{ Reserve method
    PROC Reserve(CHAN OF SP fs, ts, [] BYTE fl, [] BYTE ag, [] BYTE d,
        INT len, [] BYTE name,
        CHAN OF ACK to.interface)

SEQ
    found.sched := FALSE
    WHILE NOT found.sched
        SEQ
            flight := [Res.table[i] FROM 0 FOR 3]
            agent := [Res.table[i] FROM 3 FOR 3]
            date := [Res.table[i] FROM 6 FOR 6]
            key := [Res.table[i] FROM 12 FOR 4]
            IF
                eqstr(flight, fl) AND eqstr(date, d)
            --{{
                check if booking does not already exist and book
            --}}
            SEQ
                found.name := FALSE
                j := 0
                WHILE NOT found.name
                    SEQ
                        IF
                            eqstr([key FROM 0 FOR 4], [Passlist[j] FROM 0 FOR 4])
                                AND eqstr([name FROM 0 FOR len], [Passlist[j] FROM 4 FOR len])
                        Booking already made
                        TRUE
                        SEQ
                            IF
                                Passlist[j][0] = '@'

138
SEQ
[Passlist[j] FROM 0 FOR 4] := key
[Passlist[j] FROM 4 FOR len] :=
    [name FROM 0 FOR len]

Make booking

resp := TRUE
Npassengers := Npassengers + 1
found.name := TRUE
TRUE
j := j + 1
found.sched := TRUE
--))})

TRUE
--{( create new schedule and book passenger

IF
    resp = TRUE
    to.interface ! single.ack; resp; 1
    : ;
--))})

--{( Cancel method
PROC Cancel([] BYTE fl, [] BYTE ag, [] BYTE d, INT len, [] BYTE name,
        CHAN OF ACK to.router)

Similar to reserve method

--{( Agent reservations method
PROC Agent.reservations(CHAN OF SP fs, ts, [] BYTE fl, [] BYTE ag,
        [] BYTE d, CHAN OF ACK to.router)

SEQ
found.sched := FALSE
i := 0
WHILE NOT found.sched
SEQ
    flight := [Res.table[i] FROM 0 FOR 3]
    agent := [Res.table[i] FROM 3 FOR 3]
    date := [Res.table[i] FROM 6 FOR 6]
    key := [Res.table[i] FROM 12 FOR 4]
    IF
        eqstr(flight, fl) AND eqstr(date, d)
        --{( retrieve reservations
    SEQ
        Display agent reservation list
        j := 0
        WHILE j <= Npassengers
        IF
            eqstr([key FROM 0 FOR 4], [Passlist[j] FROM 0 FOR 4])
            SEQ
                num.pass := num.pass + 1
            }}}}
\[ j := j + 1 \]
TRUE
\[ j := j + 1 \]
IF
num.pass = 0
\textit{No reservations made}
TRUE
resp := TRUE
found.sched := TRUE
--})})
TRUE
--{{ no such flight schedule exists
--})})
to.router ! single.ack;resp;1
:
--})})
SEQ
going := TRUE
WHILE going
--{{ select method
PAR
ALT
from.trouter ? CASE

\textbf{resv}:flight.no;agent1:date:size::passenger
--{{ make reservation
SEQ
to.flight ! chk.bking:flight.no;caller
ALT
from.flight ? CASE
  single.ack;resp;caller
  IF
    resp = FALSE
    SEQ
to.trouter ! single.ack;FALSE;agl
    TRUE
    SEQ
    Reserve(fs,ts,flight.no,agent1,date,size,passenger,to.trouter)
--})})
\textbf{canc}:flight.no2;agent2:date:size::passenger
--{{ cancel reservation
SEQ
  Cancel(flight.no2,agent2,date,size,passenger,to.trouter)
--})})
\textbf{ares}:flight.no;agent3:date
--{{ display agent reservations
SEQ
  Agent.reservations(fs,ts,flight.no,agent3,date,to.trouter)
  to.trouter ! single.ack;TRUE;ag3
--})})
f) **Fares Class**

PROC Fares(CHAN OF SP fs, ts, CHAN OF FAREPRO from.mux, CHAN OF ACK to.mux, CHAN OF INITPRO init)

```plaintext
---{} Change fare method
PROC ChangeFare(CHAN OF SP f, t, []BYTE fl.no, INT fare, ctype, caller, CHAN OF ACK msg)
SEQ
    finish := FALSE
    i := 0
    newfare := 0
    WHILE NOT finish
        IF
            eqstr([Fares.tab[i] FROM 0 FOR 3], fl.no)
            SEQ
                STRINGINT(err, cfare, [Fares.tab[i] FROM 3 FOR 5])
                IF
                    ctype = 1
                    newfare := cfare + fare
                    ctype = 2
                    newfare := cfare - fare
                    TRUE
                    SKIP
                INTTOSTRING(len, strfare, newfare)
                [Fares.tab[i] FROM 3 FOR 5] := [strfare FROM 0 FOR 5]
                IF
                    ctype = 1
                    SEQ
                        Display message about fare increase
                    TRUE
                    SEQ
                        Display message about fare decrease
                resp := TRUE
                finish := TRUE
        TRUE
        IF
            i = Nfares
            Cannot change fare.
            TRUE
            i := i + 1
        msg ! single.ack; resp; caller
    :
---{} Display fare method
PROC DisplayFare(CHAN OF SP f, t, []BYTE fl.no, INT caller, CHAN OF ACK msg)
SEQ
    finish := FALSE
    i := 0
    WHILE NOT finish
        IF
            eqstr([Fares.tab[i] FROM 0 FOR 3], fl.no)
```
SEQ

Display details of fares.
TRUE
SEQ
IF
    i = Nfares
    Flight does not exist
TRUE
    i := i + 1
msg ! single.ack;TRUE;caller

--{}--

Add new fare method
PROC AddNewFare(CHAN OF SP f,t,[],BYTE f1.no,INT fare,caller,
CHAN OF ACK msg)

WHILE NOT finish
IF
    eqstr(f1.no,[Fares.tab[i] FROM 0 FOR 3]) OR
      (Fares.tab[i][0] = '8')
SEQ
    INTTOSTRING(len,strfare,fare)
    [Fares.tab[i] FROM 0 FOR 3] := f1.no
    [Fares.tab[i] FROM 3 FOR 5] := [strfare FROM 0 FOR 5]
    Nfares := Nfares + 1
    resp := TRUE
    finish := TRUE
TRUE
IF
    i = FLMAX
    Flight no found
TRUE
    i := i + 1
msg ! single.ack;resp;caller

:--{}--

SEQ
WHILE TRUE
ALT
from.mux ? CASE
  chng.fare:flight.nol:newfare1:chngtype:whol
    ChangeFare(fs,ts,flight.nol,newfare1,chngtype,whol,to.mux)
disp.fare:flight.nol:who2
    DisplayFare(fs,ts,flight.nol,who2,to.mux)
new.fare:flight.nol:newfare2:who3
    AddNewFare(fs,ts,flight.nol,newfare2,who3,to.mux)

C2. MULTIPLEXORS AND DEMULTIPLEXORS

The following are two examples of multiplexors and demultiplexors used.
a) This object multiplexes messages and sends them to instances of class Tragent.
PROC Q.to.Tmux(CHAN OF RAWDATA from.query, CHAN OF NAGENT to.procm an, from.procm an,[ ]CHAN OF RAWDATA to.tragent)

INT agent,agent1,len:
[40]BYTE query:
BOOL active,ok:
INT id,agent.num:
SEQ
active := TRUE

PAR
WHILE active (activates multiplexor)
SEQ
ALT
from.query ? CASE
dat;len::query:id
SKIP
to.procm an ! num.agents;0 (checks number of active instances of Travel agents)

ALT
from.procm an ? CASE
num.agents;agent.num
agent := id REM agent.num
to.tragent[agent] ! dat;len::query;agent (multiplex and send)

--------------------------------------------------------------------------

a) This object receives messages from Tragent and then de multiplexes them and sends them to Trquery.

PROC Tr.to.Qmux([ ]CHAN OF ACK from.tragent, CHAN OF ACK to.trquery)

BOOL ok:
INT agent:

ALT i = 0 FOR 4
from.tragent[i] ? CASE (receive message)
single.ack;ok;agent
to.trquery ! single.ack;ok;agent (demultiplex message and send)

--------------------------------------------------------------------------

C3. ROUTERS

The following is an example of a router. It routes incoming flight queries.

PROC Route.to.flight([ ]CHAN OF FLIGHTPRO flight.req,
CHAN OF FLIGHTPRO to.oflight,to.mflight)
SEQ
WHILE TRUE

ALT i = 0 FOR 3
flight.req[i] ? CASE

find.f1;f1.nol;any
-{{ route find flight
IF
    f1.nol[0] = 'O'
    to.oflight ! find.f1;f1.nol;i
    TRUE
    to.mflight ! find.f1;f1.nol;i
-}}}

add.f1;f1.no2;capl;bookedl;numlegs1;any
-{{ route add flight
IF
    f1.no2[0] = 'O'
    to.oflight ! add.f1;f1.no2;capl;bookedl;numlegs1;i
    TRUE
    to.mflight ! add.f1;f1.no2;capl;bookedl;numlegs1;i
-}}}

drop.f1;f1.no3;any
-{{ route drop flight
IF
    f1.no3[0] = 'O'
    to.oflight ! drop.f1;f1.no3;i
    TRUE
    to.mflight ! drop.f1;f1.no3;i
-}}}

chk.bking;f1.no4;any
-{{ route check booking
IF
    f1.no4[0] = 'O'
    to.oflight ! chk.bking;f1.no4;i
    TRUE
    to.mflight ! chk.bking;f1.no4;i
-}}}

disp.f1;f1.no5;any
-{{ route display flight
IF
    f1.no5[0] = 'O'
    to.oflight ! disp.f1;f1.no5;i
    TRUE
    to.mflight ! disp.f1;f1.no5;i
-}}}

C4. PROTOCOLS

This appendix shows the structures of some of the class protocols used.

a) Protocol for acknowledgment used by all classes to acknowledge query.
b) Protocol for class Amparse.

PROTOCOL AMPRO
CASE
tf:[3]BYTE;INT -- [find fl] fl.no, manager
af:[3]BYTE;INT;[3]BYTE;INT;[3]BYTE -- [add fl] fl.no,
    manager, cap, booked, numlegs
df:[3]BYTE;INT -- [drop flight] fl.no, manager
nf:[3]BYTE;INT;[5]BYTE -- [new fare] fl.no, manager, new fare
yf:[3]BYTE;INT -- [show flight] fl.no, manager
zf:[3]BYTE;INT -- [show fare] fl.no, manager
al:[3]BYTE;INT;[5]BYTE -- [add a leg] fl.no, manager, fare increase
dl:[3]BYTE;INT;[5]BYTE -- [drop a leg] fl.no, manager, fare decrease
ane;INT::[5]BYTE;INT;BOOL
ade;INT::[5]BYTE;INT;BOOL


c) Protocol for class Fares.

PROTOCOL FAREPRO
CASE
chg.fare:[3]BYTE;INT;INT;INT -- fl.no, fare, changetype, caller.id
disp.fare:[3]BYTE;INT -- fl.no, caller.id
new.fare:[3]BYTE;INT;INT -- fl.no, fare, caller.id


d) Protocol for the travel agent object manager Tr_objman.

PROTOCOL NAGENT
CASE
curr.agents
num.agents;INT
new;INT::[5]BYTE;INT -- new agent name, identity of caller object
del;INT::[5]BYTE;INT -- agent t be deleted, " "


e) Protocol for all class members of the Flight inheritance hierarchy.

PROTOCOL FLIGHTPRO
CASE
find.fl:[3]BYTE;INT -- flt.no, caller.id
add.fl:[3]BYTE;[3]BYTE;INT;[3]BYTE;INT -- flt.no, cap, booked, nlegs, id
drop.fl:[3]BYTE;INT -- flt.no, caller.id
disp.fl:[3]BYTE;INT -- flt.no, caller.id
f) Protocol for class Parser.

PROTOCOI PARSEP
CASE
  pr.fll[3]BYTE
  pr.datlINT::[]BYTE
  isit.ok;BOOL
:

C5. SUBSYSTEMS.

The structure of cluster of classes or subsystems for distribution to 4 processors.

PROC Arsystem(CHAN OF SP fs, ts, CHAN OF RAWDATA airmanquery, tragentquery,
CHAN OF AMPRO aresp, CHAN OF TRAPRO tresp)

|--{ {{
|  include files
|    Include all class protocols
|    Include all compiled classes.
|--})

|--{ { channels
|--})

PAR
|--{ { run flight, fare and reserve objects

  so.multiplexor(fs, ts, from.proc, to.proc, stop)
SEQ
WHILE TRUE
PAR
  Init.arrays(init)
  AirmanQuery(to.proc[0], from.proc[0], airmanquery, from.ainterface)
  TragentQuery(to.proc[1], from.proc[1], tragentquery,
               from.tinterface)
  Am.Interface(to.proc[6], from.proc[6], aresp, from.ainterface,
               to.farelux1[0], from.farelux2[0], to.flinkux1[0],
               from.flinkux2[0])
  Tr.Interface(to.proc[7], from.proc[7], tresp, from.tinterface,
               to.farelux1[1], to.flinkux1[1], from.faremux2[1], from.felmux2[1],
               from.reserve, to.reserve)
  Mux.to.flight(to.flinkux1, oflight.req, mflight.req)
Mux.from.flight(from.cflight,from.flmux2)
One.stop.flight(from.flight[1],to.flight[1],oflight.req,
    from.cflight[0])
Multi.stop.flight(mflight.req,from.cflight[1],to.flimage[0],
    to.flight[0],from.flimage[0],from.flight[0])
Flight(from.flight,to.flight,to.flimage[1],from.flimage[1])
Flight.image(to.proc[3],from.proc[3],to.faremux1[2],
    from.faremux2[2],to.flimage,from.flimage,
    init[1])
Reservation(to.proc[4],from.proc[4],to.flmux1[2],from.flmux2[2],
    from.reserve,to.reserve,init[0])
Mux.to.Fare(to.faremux1,to.fare)
Mux.from.Fare(from.fare,from.faremux2)
Fares(to.proc[5],from.proc[5],to.fare,from.fare,init[2])

stop ! FALSE
--})

PROC Gparser (CHAN OF TRAPRO tparse,CHAN OF AMPRO aparse,
    CHAN OF ACK aparsack,tparsack)

Include files
[2]CHAN OF PARSEP to.parse:
[2]CHAN OF PARSEP from.parse:

WHILE TRUE
SEQ
  --so.write.string.nl(fs,ts,"in parser")
  --{{ run parser objects
  PAR
    AmParseQuery(aparse,aparsack,to.parse[1],from.parse[1])
    Parse.item(to.parse,from.parse)
    TrParseQuery(tparse,tparsack,to.parse[0],from.parse[0])
  --}}

PROC Amanager (CHAN OF RAWDATA query,CHAN OF AMPRO aparse,
    CHAN OF ACK parsack,CHAN OF AMPRO aresp)

Include class protocols
Include compiled classes

--{{ CHANNELS
--}}}
WHILE TRUE
SEQ
--{}  run airline manager objects
PAR
Init.manager(init)
Q.to.Amux(query,to.aprocmannl[0],from.aprocmannl,from.amquery)
Am.to.Qmux(to.amquery,aresp)
Am.proc.manager(to.aprocmannl,from.aprocmannl,to.airman2,init)
Am.to.procmux(to.aprocmannl,to.aprocmannl[1])
Aproc.amux(to.airman2,from.aprocmannl)

PAR j = 0 FOR 4
AirlineManager(to.aprocmannl[j],from.aprocmannl[j],to.amparse[j],from.amparse[j],from.amquery[j],to.amquery[j])
Am.to.Pmux(to.amparse,aparse)
P.to.Amux(parsack,from.amparse)
--{}  }

PROC Travelag(CHAN OF RAWDATA query,CHAN OF TRAPRO tparse,
CHAN OF ACK parsack,CHAN OF TRAPRO tresp)

Include class protocols
Include compiled classes

--{}  
--})

SEQ
WHILE TRUE
--{}  run travel agent objects
PAR
Init.travelagen(init)
Q.to.Trmux(query,to.tprocman[0],from.tprocman,from.trquery)
Tr.to.Qmux(to.trquery,tresp)
Tr.proc.manager(to.tprocman,from.tprocman,to.travelagen,init)
Tr.to.procmux(to.tprocmanl,to.tprocmanl[1])
Tproc.trmux(to.travelagen,from.tprocmanl)
PAR i = 0 FOR 4
TravelAgent(from.airman3[i],
to.airman3[i],from.trquery[i],to.trquery[i],
to.trparse[i],from.trparse[i],to.tprocmanl[i],
from.tprocmanl[i])
Tr.to.Pmux(to.trparse,tparse)
P.to.Trmux(parsack,from.trparse)
--{}  }

C6. CONFIGURATION DESCRIPTIONS

The following are outlines of 2 configuration descriptions.

a) Configuration description of system to run on the root transputer.

--{}  include files

148
b) Configuration description of system to run on three transputers.

```plaintext
--{{
  include files
  Include class protocols
  Include linked subsystems
}}

--{{
  internode channels

CHAN OF SP fs, ts:
CHAN OF AMPRO aparse, amresp:
CHAN OF TRAPRO tpars, tresp:
CHAN OF RAWDATA amquery, tquery:
CHAN OF ACK amparsack, tparsack:
}}

--{{
  configuration description

PLACED PAR

PROCESSOR 0 T800

--{{
  EXTERNAL CHANNELS

PLACE ts AT link0.out:
PLACE fs AT link0.in:
}}

CHAN OF AMPRO aparse, aresp:
CHAN OF TRAPRO tpars, tresp:
CHAN OF RAWDATA aquery, tquery:
CHAN OF ACK aparsack, tparsack:

PAR
  Arsysteem(fs, ts, aquery, tquery, aresp, tresp)
  Amanager(aquery, aparse, aparsack, aresp)
  Travelag(tquery, tpars, tparsack, tresp)
  Gparser(tpars, aparse, aparsack, tparsack)
}}
```
PLACE amquery AT link3.out:
PLACE amresp AT link3.in:
--{}))

PAR
Arsystem(fs, ts, amquery, trquery, amresp, tresp)

PROCESSOR 1 T800

--{{ EXTERNAL CHANNELS:
PLACE trquery AT link0.in:
PLACE tresp AT link0.out:
PLACE trparsack AT link1.in:
PLACE trparse AT link1.out:

--{}))

PAR
Travelag(trquery, trparse, trparsack, tresp)

PROCESSOR 3 T800

PLACE trparse AT link0.in:
PLACE trparsack AT link0.out:
PLACE amquery AT link3.in:
PLACE amresp AT link3.out:
CHAN OF AMPRO aparse:
CHAN OF ACK aparsack:

PAR
Gparser(trparse, aparse, aparsack, trparsack)
Amanager(amquery, aparse, aparsack, amresp)

--{}))

-----------------------------------------------------------------------
APPENDIX D. The Hardware.

D1. The IMS T800 Transputer Architecture.

**IMS T800 transputer**

**Features**
- 32-bit architecture
- 35 ns internal cycle time
- 30 MIPS (peak) instruction rate
- 4.3 MFlops (peak) instruction rate
- 364-bit on-chip floating point unit which conforms to
  IEEE 754
- 1 Kbytes on-chip static RAM
- 120 Mbytes/sec sustained data rate to internal memory
- 4 Gbytes directly addressable external memory
- 40 Mbytes/sec sustained data rate to external memory
- 330 ns response to interrupts
- Four INMOS serial links 5/10/20 Mbits/sec
- Bi-directional data rate of 2.4 Mbytes/sec per link
- High performance graphics support with block move
  instructions
- Boot from ROM or communication links
- Single 5 MHz clock input
- Single ±5V ±5% power supply
- MIL-STD-883C processing is available

**Applications**
- Scientific and mathematical applications
- High speed multi processor systems
- High performance graphics processing
- Supercomputers
- Workstations and workstation clusters
- Digital signal processing
- Accelerator processors
- Distributed databases
- System simulation
- Telecommunications
- Robotics
- Fault tolerant systems
- Image processing
- Pattern recognition
- Artificial intelligence
APPENDIX E. Testing

We show below in E1 some typical queries for an airline manager and their subsequent executions on a network of three transputers in E2. Host communication is included.

E1. A list of Airline Manager Queries.

af;MF9;500;007
nf;MF5;00700
yf;MF5
zf;MF5
al;MF2;00040
dl;MF5;00040

E2. Execution of queries in E1.

Added flight -MF9
Duration of Mngr query number 13 is - 268

new fare for flight - MF5 $700
Duration of Mngr query number 23 is - 95

Details of Flight : MF5
Capacity : 100
Number of bookings : 0
Number of Stops : 10
Duration of Mngr query number 33 is - 370

FARES FOR FLIGHT : MF5
--------------------------
Economy : $1400
Business Class : $2100
First Class : $2800
Children < 10 : $700
Duration of Mngr query number 43 is - 423

Leg added to flight MF2 fare increased to $440
Duration of Mngr query number 47 is - 187

Leg dropped from flight OF4 fare decreased to : $110
Duration of Mngr query number 48 is - 188

153
D2. Transputer configuration

The TREE configuration for the Transputer network.
Comparison of Managers Queries running on the host, 1 transputer and 3 transputers with host communication included. Timings are in ticks.

<table>
<thead>
<tr>
<th>Qr_No.</th>
<th>Host</th>
<th>Root-T</th>
<th>3-Transputers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1991</td>
<td>1979</td>
<td>1972</td>
</tr>
<tr>
<td>1</td>
<td>279</td>
<td>270</td>
<td>262</td>
</tr>
<tr>
<td>2</td>
<td>308</td>
<td>296</td>
<td>286</td>
</tr>
<tr>
<td>3</td>
<td>311</td>
<td>299</td>
<td>289</td>
</tr>
<tr>
<td>4</td>
<td>310</td>
<td>298</td>
<td>288</td>
</tr>
<tr>
<td>5</td>
<td>315</td>
<td>304</td>
<td>293</td>
</tr>
<tr>
<td>6</td>
<td>289</td>
<td>279</td>
<td>268</td>
</tr>
<tr>
<td>7</td>
<td>281</td>
<td>270</td>
<td>261</td>
</tr>
<tr>
<td>8</td>
<td>280</td>
<td>268</td>
<td>260</td>
</tr>
<tr>
<td>9</td>
<td>147</td>
<td>140</td>
<td>135</td>
</tr>
<tr>
<td>10</td>
<td>151</td>
<td>144</td>
<td>138</td>
</tr>
<tr>
<td>11</td>
<td>286</td>
<td>276</td>
<td>267</td>
</tr>
<tr>
<td>12</td>
<td>289</td>
<td>277</td>
<td>266</td>
</tr>
<tr>
<td>13</td>
<td>289</td>
<td>277</td>
<td>268</td>
</tr>
<tr>
<td>14</td>
<td>279</td>
<td>269</td>
<td>260</td>
</tr>
<tr>
<td>15</td>
<td>152</td>
<td>145</td>
<td>141</td>
</tr>
<tr>
<td>16</td>
<td>127</td>
<td>123</td>
<td>118</td>
</tr>
<tr>
<td>17</td>
<td>147</td>
<td>140</td>
<td>138</td>
</tr>
<tr>
<td>18</td>
<td>99</td>
<td>96</td>
<td>94</td>
</tr>
<tr>
<td>19</td>
<td>99</td>
<td>96</td>
<td>94</td>
</tr>
<tr>
<td>20</td>
<td>99</td>
<td>96</td>
<td>94</td>
</tr>
<tr>
<td>21</td>
<td>99</td>
<td>96</td>
<td>95</td>
</tr>
<tr>
<td>22</td>
<td>99</td>
<td>96</td>
<td>95</td>
</tr>
<tr>
<td>23</td>
<td>100</td>
<td>97</td>
<td>95</td>
</tr>
<tr>
<td>24</td>
<td>419</td>
<td>405</td>
<td>396</td>
</tr>
<tr>
<td>25</td>
<td>817</td>
<td>718</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>421</td>
<td>406</td>
<td>395</td>
</tr>
<tr>
<td>27</td>
<td>422</td>
<td>407</td>
<td>395</td>
</tr>
<tr>
<td>28</td>
<td>393</td>
<td>379</td>
<td>371</td>
</tr>
<tr>
<td>29</td>
<td>392</td>
<td>380</td>
<td>369</td>
</tr>
<tr>
<td>30</td>
<td>394</td>
<td>380</td>
<td>369</td>
</tr>
<tr>
<td>31</td>
<td>395</td>
<td>380</td>
<td>371</td>
</tr>
<tr>
<td>32</td>
<td>395</td>
<td>380</td>
<td>370</td>
</tr>
<tr>
<td>33</td>
<td>469</td>
<td>453</td>
<td>446</td>
</tr>
<tr>
<td>34</td>
<td>470</td>
<td>453</td>
<td>448</td>
</tr>
<tr>
<td>35</td>
<td>441</td>
<td>427</td>
<td>421</td>
</tr>
<tr>
<td>36</td>
<td>837</td>
<td>527</td>
<td>620</td>
</tr>
<tr>
<td>37</td>
<td>443</td>
<td>426</td>
<td>420</td>
</tr>
<tr>
<td>38</td>
<td>679</td>
<td>700</td>
<td>713</td>
</tr>
<tr>
<td>39</td>
<td>444</td>
<td>428</td>
<td>422</td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparison of Managers Queries running on the host, 1 transputer and 3 transputers without host communication. Timings are in ticks.

<table>
<thead>
<tr>
<th>Qr No.</th>
<th>Host</th>
<th>Root-T</th>
<th>3-Transputers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1759</td>
<td>1734</td>
<td>1731</td>
</tr>
<tr>
<td>1</td>
<td>17</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>17</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>17</td>
<td>17</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>18</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>18</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>18</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>18</td>
<td>18</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>19</td>
<td>19</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>20</td>
<td>20</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>21</td>
<td>19</td>
<td>17</td>
</tr>
<tr>
<td>11</td>
<td>21</td>
<td>20</td>
<td>17</td>
</tr>
<tr>
<td>12</td>
<td>21</td>
<td>20</td>
<td>17</td>
</tr>
<tr>
<td>13</td>
<td>22</td>
<td>21</td>
<td>18</td>
</tr>
<tr>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>15</td>
<td>13</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>17</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>18</td>
<td>15</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>19</td>
<td>14</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>15</td>
<td>13</td>
</tr>
<tr>
<td>22</td>
<td>15</td>
<td>14</td>
<td>13</td>
</tr>
<tr>
<td>23</td>
<td>15</td>
<td>15</td>
<td>14</td>
</tr>
<tr>
<td>24</td>
<td>16</td>
<td>15</td>
<td>12</td>
</tr>
<tr>
<td>25</td>
<td>16</td>
<td>15</td>
<td>12</td>
</tr>
<tr>
<td>26</td>
<td>15</td>
<td>15</td>
<td>13</td>
</tr>
<tr>
<td>27</td>
<td>15</td>
<td>15</td>
<td>13</td>
</tr>
<tr>
<td>28</td>
<td>15</td>
<td>16</td>
<td>13</td>
</tr>
<tr>
<td>29</td>
<td>15</td>
<td>15</td>
<td>12</td>
</tr>
<tr>
<td>30</td>
<td>15</td>
<td>15</td>
<td>13</td>
</tr>
<tr>
<td>31</td>
<td>17</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>32</td>
<td>17</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>33</td>
<td>17</td>
<td>17</td>
<td>13</td>
</tr>
<tr>
<td>34</td>
<td>9</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>35</td>
<td>10</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>36</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>37</td>
<td>10</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>38</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>39</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>40</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>
Comparison of Travelagent Queries running on the host, 1, 3 and 4 transputers with host communication included. Timings are in ticks.

<table>
<thead>
<tr>
<th>Query No.</th>
<th>Host</th>
<th>1-trans</th>
<th>3-trans</th>
<th>4-trans</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>547</td>
<td>527</td>
<td>514</td>
<td>514</td>
</tr>
<tr>
<td>2</td>
<td>574</td>
<td>555</td>
<td>540</td>
<td>540</td>
</tr>
<tr>
<td>3</td>
<td>578</td>
<td>554</td>
<td>540</td>
<td>540</td>
</tr>
<tr>
<td>4</td>
<td>579</td>
<td>557</td>
<td>541</td>
<td>544</td>
</tr>
<tr>
<td>5</td>
<td>578</td>
<td>559</td>
<td>541</td>
<td>542</td>
</tr>
<tr>
<td>6</td>
<td>581</td>
<td>560</td>
<td>544</td>
<td>545</td>
</tr>
<tr>
<td>7</td>
<td>584</td>
<td>561</td>
<td>545</td>
<td>544</td>
</tr>
<tr>
<td>8</td>
<td>279</td>
<td>266</td>
<td>264</td>
<td>263</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>455</td>
<td>434</td>
<td>427</td>
<td>425</td>
</tr>
<tr>
<td>11</td>
<td>455</td>
<td>434</td>
<td>427</td>
<td>427</td>
</tr>
<tr>
<td>12</td>
<td>523</td>
<td>501</td>
<td>428</td>
<td>427</td>
</tr>
<tr>
<td>13</td>
<td>453</td>
<td>434</td>
<td>427</td>
<td>427</td>
</tr>
<tr>
<td>14</td>
<td>239</td>
<td>231</td>
<td>222</td>
<td>223</td>
</tr>
<tr>
<td>15</td>
<td>413</td>
<td>95</td>
<td>94</td>
<td>95</td>
</tr>
<tr>
<td>16</td>
<td>274</td>
<td>262</td>
<td>256</td>
<td>254</td>
</tr>
<tr>
<td>17</td>
<td>91</td>
<td>87</td>
<td>85</td>
<td>85</td>
</tr>
<tr>
<td>18</td>
<td>243</td>
<td>236</td>
<td>213</td>
<td>214</td>
</tr>
<tr>
<td>19</td>
<td>405</td>
<td>387</td>
<td>587</td>
<td>574</td>
</tr>
<tr>
<td>20</td>
<td>239</td>
<td>230</td>
<td>222</td>
<td>222</td>
</tr>
</tbody>
</table>