THE EFFECTIVE USE OF IMPLICIT PARALLELISM THROUGH THE USE OF AN OBJECT-ORIENTED PROGRAMMING LANGUAGE

DAVID RANN

Doctor of Philosophy

THE UNIVERSITY OF ASTON IN BIRMINGHAM

February 1996

The University of Aston in Birmingham

THE EFFECTIVE USE OF IMPLICIT PARALLELISM THROUGH THE USE OF AN OBJECT-ORIENTED PROGRAMMING LANGUAGE

DAVID RANN

Doctor of Philosophy
1996

Thesis Synopsis

This thesis explores translating well-written sequential programs in a subset of the Eiffel programming language - without syntactic or semantic extensions - into parallelised programs for execution on a distributed architecture. The main focus is on constructing two object-oriented models: a theoretical self-contained model of concurrency which enables a simplified second model for implementing the compiling process. There is a further presentation of principles that, if followed, maximise the potential levels of parallelism.

Model of Concurrency

The concurrency model is designed to be a straightforward target for mapping sequential programs onto, thus making them parallel. It aids the compilation process by providing a high level of abstraction, including a useful model of parallel behaviour which enables easy incorporation of message interchange, locking, and synchronization of objects. Further, the model is sufficient such that a compiler can and has been practically built.

Model of Compilation

The compilation-model's structure is based upon an object-oriented view of grammar descriptions and capitalises on both a recursive-descent style of processing and abstract syntax trees to perform the parsing. A composite-object view with an attribute grammar style of processing is used to extract sufficient semantic information for the parallelisation (i.e. code-generation) phase.

Programming Principles

The set of principles presented are based upon information hiding, sharing and containment of objects and the dividing up of methods on the basis of a command/query division. When followed, the level of potential parallelism within the presented concurrency model is maximised. Further, these principles naturally arise from good programming practice.

Summary

In summary this thesis shows that it is possible to compile well-written programs, written in a subset of Eiffel, into parallel programs without any syntactic additions or semantic alterations to Eiffel: i.e. no parallel primitives are added, and the parallel program is modelled to execute with equivalent semantics to the sequential version. If the programming principles are followed, a parallelised program achieves the maximum level of potential parallelisation within the concurrency model.

KEYWORDS: compiler, parallelisation, deadlock, Concurrent Object Machine

Contents

List of Figures

Part I

Introduction

Introduction to part I

Part one introduces the basic thesis upon which this document is based. Chapter 1 presents the thesis and includes a discussion of the perceived problem and a possible solution. Chapters 2, 3 and 4 present the theoretical background material. Chapter 5, given the previous chapters as a base, presents the thesis objectives.

Chapter 1

Introduction

1.1 Object-oriented Parallelism: a Solution?

In computing there is an ever-increasing quest for more powerful computers which can be used to make existing software execute faster or provide sufficient power to execute more complex pieces of software that are impractical on current hardware platforms. This quest however has boundaries based upon the current level of technology and processor performance at any time. However, if a number of processors are combined to work upon a problem the outcome should be a generally more powerful computer. This increase in power is only noticeable if a problem is successfully partitioned into a number of subprograms which can be placed upon different processors and coordinated towards the production of a solution. It is this partitioning and the consequent management of the executing parts that gives rise to the problems of parallelism that many programmers find hard. Therefore an ideal aim would be the full automation of the parallelisation[1] of sequential programs suitable for execution on a parallel computer.

It is suggested by the title and throughout this thesis that object-oriented modelling is a useful technique for tackling this problem and that it is possible to fulfil the total automation of the translation process. It might be reasonably asked

1. why automatic parallelisation?

2. why object-oriented modelling?

The rest of this chapter and indeed the rest of the thesis address these two questions.

[1] "Parallelisation" is used throughout the thesis to mean the translating of a sequential description of an algorithm or program into an equivalent parallel version.

1.1.1 Why Automatic Parallelisation?

One of the main justifications for this work is the difficulty involved in the programming of parallel systems. Even for "capable" programmers the harnessing of multiple processors is not easy. One of the main problems is that extra levels of difficulty are involved in the visualisation of the processes of computation, particularly in comparison with the development of sequential programs. Consequently, there is a greater possibility for the introduction of errors.

For example, consider the problem of memory management: some programmers need to deal explicitly with memory management, thinking about what portion of memory to use for a particular task, or what is the exact size of a particular data structure. They may need to keep explicit track of memory usage. However, many programmers these days, using their C or C++ programming skills and the operating system upon which they are developing, only need to consider *"new-ing"* and *"free-ing"* areas of memory and not explicit locations. Memory management is easier still for the normal Eiffel or Smalltalk programmer. They can forget all about such issues as allocation and deallocation and rely instead on the underlying language framework and garbage collector.

It would be beneficial if parallel programming could be moved along an analogous path to that of memory management. Instead of most programmers needing to explicitly manage processes, they should be able to leave it to the compiler and the operating system in use. Some programmers will always need to state explicitly that process x must go on processor b and thread j must have priority over thread k. However, for the vast majority of applications, this precise level of control is neither necessary nor warranted.

1.1.2 Why Object-Oriented Modelling?

The "fundamental model" can help or hinder implementation, enhancement and efficiency (a good model may not necessarily be good in terms of speed) (Brunskill, Rann & Turner 1995). Consequently, the modelling technique chosen must enable effective reasoning about what ultimately is a complex system.

Given my background in formal techniques (Rann, Turner & Whitworth 1994) the first approach considered was that of "formality". Formal models, whilst very precise, are problematic in this domain. The approach of CSP and CCS with their fundamental operators leads to a description technique which seems unsuited to systems as inherently complex as that described in this thesis. Equally a Z or VDM based approach, whilst effective at describing various entities, is not an effective approach for describing concurrent object-oriented systems. There are other formal modelling techniques - Process algebras, Petri Nets etc - but

fundamentally the abstraction mechanisms supported are too detailed for useful reasoning and modelling. Therefore a formal approach, in the mathematical sense, was not deemed practical, as deriving the vital set of axioms or constructive mathematical model would be too time-consuming and would move the research objectives away from the parallelisation of sequential programs, to a mathematical modelling of object-oriented concurrency.

The modelling technique chosen needed to support a variety of levels of reasoning, to help in the handling of the complexity inherent in the system. Alan Kay once stated in a seminar[2] that there are three main approaches to reasoning. The first and simplest taken by a young child is that of *mimicry,* where a child achieves a purpose by acting it out - e.g. drawing a circle. A young child can think of the process involved by standing at a point and then taking a step and turning a bit and then another step and turning a bit etc, until a circle is drawn. A slightly older child can take an approach based upon *images.* They know that a circle is drawn with all its points equidistant from some centre, so they stand in the middle and walk to the outside draw a point come back to the centre turn a bit, go to the outside draw a point and then come back to the centre etc, until the circle is drawn. The final stage, and not one achieved by everybody is the *symbolic* level. For example a teenager may know that a circle is described by $r^2 = x^2 + y^2$ and may also know that this can be plotted and how. These three levels of reasoning - *mimcry, imagery, symbolism* - are each effective in their own right, but the ones attained by the older children become increasingly more powerful as tools for manipulation and succinct reasoning, hence my original wish to use mathematical modelling.

Object-oriented software development supports the aforementioned three levels of reasoning. In this work's design of a system to support concurrency, a lot of emphasis has been placed on the soundness of the model and overloading of ideas and names so that others can pick this up and hopefully read and comprehend it. Refinements of this model at a later date would enable efficiency gains, but refinement of a initial model which, for whatever reason, is weak, would be much more difficult. Hence the earlier aim to show the reasonableness of the approach and not necessarily its immediate effectiveness with respect to speed of code and optimum usage of given hardware.

Therefore, in trying to derive/produce an effective model of concurrency into which an object-oriented language can be mapped, all three levels of reasoning are called upon. In places this gives rise to naming conventions that may initially be thought pretentious or misleading, but they have been deliberately chosen for the purpose of helping in the visualisation of the system and the approach used.

[2]from The Distinguished Lecture Series: Doing with Images makes Symbols: Communicating with Computers

1.2 The Problem and a Solution

1.2.1 Summary of the Problem

The basic idea which underlies the investigation of the application of object-oriented techniques to the automatic parallelisation of sequential programs can be summarised as follows:

> It should be possible to write a program in a "good" object-oriented programming language, ignoring issues resulting from a possibility of execution in a parallel environment. The derivation of this concurrency should instead be automatic, freeing a programmer to concentrate upon problem solving instead of the difficulties of implementing parallel solutions. The programs written in this language, now able to utilise a parallel architecture, should be easier to write and less prone to errors, because of the decrease in complexity when compared with constructing a parallel solution. The automatic parallelisation of programs should inevitably lead to improved programmer productivity for multi-processor architectures.

1.2.2 Approach

The approach taken to the above problem in this thesis is to construct two theoretical models: a theoretical self-contained model of concurrency which will enable a simplified second model for implementing the compilation process. Also consideration will be given to the style of programming that should be applied; this style is summarised by programming principles which, if followed, will maximise the potential levels of parallelism.

The concurrency model will be designed to be a straightforward target onto which to map sequential programs, thus making them parallel. It will be expected to aid the compilation process by providing a high level of abstraction, including a useful model of parallel behaviour which enables easy incorporation of message interchange, locking, and synchronization of objects. Further, the model will be expected to be sufficient such that a compiler can be practically built.

The compilation-model's structure will be based upon an object-oriented view of grammar descriptions and will capitalise on both a recursive-descent style of processing and abstract syntax trees to perform the parsing. An alternative composite-object view with an attribute grammar style of processing will be used to extract sufficient semantic information for the parallelisation (i.e. code-generation) phase.

The set of programming principles should arise naturally from some widely accepted precepts of 'good' programming practice (see section 5.2); they will be based upon information hiding, sharing and containment of objects and the division of methods on a command/query

basis.

The programs to be parallelised will be written in a subset of Eiffel. A parallelised program is expected to have the same semantic effect as its sequential equivalent. The target architecture is to be a distributed computer formed from a network of UNIX workstations.

1.2.3 Summary

The compilation strategy will make use of the inherent logical structuring and message passing found in object-oriented programs. Attaining the optimum speed of execution is unlikely and not one of the aims. More efficient solutions could be produced by good programmers with machine-specific knowledge and hardware-specific concurrent programming languages. This situation is not too different from that of using general-purpose high-level programming languages on normal sequential architectures as opposed to capable low-level languages. On a sequential machine a good programmer familiar with the hardware and possessing a flexible machine-specific (usually low-level) language could produce quicker, more space-efficient code. This is not a sensible practice when developing (portable) sequential programs. Equally, explicit handling of parallelism should not be the norm in the concurrent world.

In summary the application of the above approach, within this thesis, will show that it is possible to compile well-written programs, written in a subset of Eiffel, into parallel programs without any syntactic additions or semantic alterations to Eiffel: i.e. no parallel primitives are added, and the parallel program is modelled to execute with equivalent semantics to the sequential version on the chosen target architecture: a network of UNIX workstations. If the programming principles are followed, a parallelised program will achieve the maximum level of potential parallelisation within the concurrency model.

1.3 The Thesis

Having outlined the problems and the approach to their solution, this section presents the thesis that all of this work is based upon. It also makes clear what is not involved in this work and clarifies what is. At its most general level this thesis might be expressed as:

> A model of concurrency can be derived which as a target for a compilation process will enable the automatic parallelisation of a sequential object-oriented program.

With the above expression of the thesis and the approach of section 1.2.2, it should be clearly stated that this thesis is *not* expected to

- derive a new object-oriented programming language;

- derive a concurrency model, described as a new abstract machine, which will enable any or all programmers to more easily program parallelism;
- derive a compilation model which will be totally applicable to all object-oriented programming languages.

The language which will be compiled is Eiffel, chosen specifically for its clear well-defined semantics and pure approach to object-oriented software, which is missing from languages such as C++.

The concurrent model and consequent parallel implementation will not necessarily demonstrate any performance improvements over the direct execution of the sequential version, in the current implementation; in fact, if execution were possible, it might demonstrate a degradation in performance[3] because of the inherently inefficient operating system tools used to make the demonstration of the ideas practical. However the model should be amenable to refinement in post-thesis work such that it can be made to be an effective target for the compilation process where performance gains are important. The type of parallel systems specifically considered within this thesis are those that exhibit a MIMD[4] style architecture.

The compilation model will contain techniques and ideas which are not immediately amenable to use on Smalltalk (Goldberg & Robson 1989), or CLOS (De Michael & Gabriel 1987), or on many other object-oriented programming languages. Indeed the ideas may not be amenable to use on C++, probably the most widely used object-oriented programming language. Instead it should deal with *well-written* programs (see section 5.2) in Eiffel, and even then because of *flaws* (see section 5.1.2) in the design of Eiffel's semantics, some Eiffel programs (though not those following the programming principles) will not benefit from the possible levels of parallelisation.

1.3.1 Summary and Contributions

This work will demonstrate that *a theoretical concurrent model of execution can be derived such that an automatic parallelisation process is possible enabling future execution of the parallel program upon a parallel system*, specifically one with a MIMD style architecture. The actual implementation, however, and even some of the basic ideas, will be open to refinement and improvement by the incorporation and use of industry-wide standards, tools and operating system extensions that have arisen within the past six years.

[3]Inherent with simulated parallelism on a single machine, but this is with reference to a distributed computer.

[4]Multiple instruction streams and multiple data streams, i.e. multiple independent processes (Andrews 1991, p 133).

The contributions of this thesis within this area are to demonstrate that given a good basic model of concurrency which is designed to avoid deadlock and issues such as starvation it is possible to automatically parallelise a well-defined sequential programming language. The theoretical concurrency model presented has been shown to be sufficient to make such a parallelisation possible, using a purely object-oriented approach to the modelling and implementation.

1.3.2 My thesis

This leads to the following fuller restatement of the thesis:

> "A theoretical concurrent model of execution can be derived such that an automatic parallelisation process of a useful subset of Eiffel programs, written using certain programming principles, is possible. The parallelised program and concurrent model are unlikely to be optimal, but with sufficient refinement - post thesis study - will achieve a potential performance improvement over that of a sequential system."

1.4 Document structure

This thesis is broken up into 5 main parts:

Part I Introduction

- Chapter 1 discusses the topic to be tackled including the perceived problem and a possible solution.
- Chapter 2, 3, 4 presents the necessary theoretical background to the work. This includes discussion of object-oriented concepts, parallelism and the combination of parallelism with object-oriented in the area of programming languages.
- Chapter 5 presents the objective for this thesis and outlines the approach taken.

Part II Design

This part follows the structure of a compiler from parsing and semantic analysis through to code generation. It presents and develops the two models central to this thesis, the model of concurrency and of the compilation process.

- Chapter 6 presents the issues involved in analysing an object-oriented program to extract sufficient information to produce an equivalent parallel program.

- Chapter 7 presents an abstract machine providing a sufficient base to map a concurrent object-oriented program onto to gain parallelism.
- Chapter 8 looks at concurrency issues and argues that deadlock has been designed out, without losing all the parallelism in execution.
- Chapter 9 presents the code generation phase of the translator.

The model of compilation can be followed through this part as the parsing and semantic anlysis are presented and discussed in chapter 6 with the code-generation phase discussed in chapter 9. The concurrency model is first introduced in chapter 7; it is extended within 8 by the discussion of how deadlock should be avoided, and is then further expanded within chapter 9 with a discussion of message handling (section 9.2), locking (section 9.4), and finally the implementation of early returns (section 9.5)

Part III Implementation

- Chapter 10 presents the overall implementation and issues involved in implementing the solution.

Part V Conclusions

- Chapter 11 evaluates the models of concurrency and compilation presented earlier in the thesis.
- Chapter 12 presents related work that has taken place during the time that this thesis has been under development and also looks at the contributions made within this thesis.
- Chapter 13 outlines future work to bring the ideas presented in the thesis up to a more commercial level of quality.
- Chapter 14 discusses the conclusions arising from the work within the thesis, and the main contributions made.

Chapter 2

Object-oriented Concepts

This chapter describes the terminology and ideas from the area of object-oriented which have been the basis for this work. The ideas presented are closely related to works by Meyer (1988) and Wegner (1987, 1990).

Applying object-oriented techniques and principles to the development of parallel systems requires agreement on what the term "object-oriented" means. This chapter presents a view of what this term means and some of the ideas and tools associated with it. "Object-orientation" as an idea is suffering from increasing devaluation through excessive use in the marketing of the latest piece of software. It also has subtly different meanings depending on the area of computing in which a person is working and the papers and books read in learning about the idea. The view, therefore, put forward in this chapter is not the marketing view, which tends to compound an already confusing plethora of terms and ideas. If anything the presented view is influenced by my background in software engineering; in the wish to build very large software systems with measurably high levels of quality.

"Object-oriented" as a term describes a specific technology. It results from the work done in developing the Simula-67 (Dahl & Nygaard 1966, Nygaard & Dahl 1978) programming language and later work done by Xerox in their Palo-Alto laboratories with the programming language Smalltalk (Goldberg & Robson 1989). When used to prefix other terms (programming, design, and analysis) it evokes a collection of ideas and tools.

Meyer's (1988, pp 60-62) text on constructing object-oriented systems summarises seven levels that, he believes, if met classify a programming language as object-oriented (see section 2.2).

Wegner (1990) describes object-oriented programming as a robust component-based modelling paradigm that is both effective and fundamental.

Snyder's (1987) paper on inheritance and the development of encapsulated software, de-

scribes object-oriented programming as a practical and useful programming methodology. He states that it aids the construction and reuse of software components because of its support for data abstraction, generic operations and inheritance.

Wegner (1987, 1990) in two papers presents a useful measure of the "object-orientedness" of a programming language. He classifies programming languages, which as a minimum support objects, as object-based, class-based, or object-oriented.

Object-based programming languages support implementation of an object-based design. Objects in this context combine encapsulated data structures with routines for affecting the data. Object-based languages support objects and their function (see section 2.1), but not their higher-order manipulation.

Class-based languages form a subset of the object-based languages; they require all objects to belong to a class. These languages not only support objects, but also their manipulation and management.

Object-oriented languages are a subset of the class-based languages; these languages support class inheritance. Class inheritance enables the language to manipulate and manage classes.

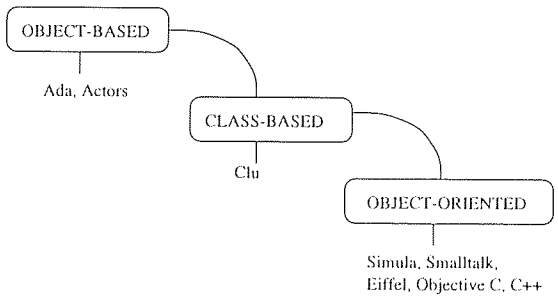


Figure 2.1: Object-based, Class-based and Object-oriented languages

As figure 2.1 (Wegner 1990, with additions) indicates, languages classed as object-oriented include Smalltalk, Simula-67 and according to Wegner's definitions languages such as Eiffel, C++, and Objective C.

Object-orientation is not just about programming languages and their features, it is a total way of viewing and developing systems from initiation to delivery and maintenance. Section 2.3 describes the building blocks of the object-oriented paradigm; it follows the structure of Meyer's (1988, chapter 4) "Seven Steps Towards Object-based Happiness", presented in section 2.2. However - given the centrality of the idea of objects - a definition of objects is presented first.

2.1 Objects

Objects are central to any object-oriented system; it is around them that any approach to, or definition of, object-oriented must be built. Therefore a working-definition of this term is presented here, and refined in section 2.3.7. The three quotations below summarise the main points of view on what an object is.

> ' "OBJECT" is a run-time notion; any object is an instance of a certain class, created at execution time and made of a number of fields.' (Meyer 1988, page 76)

2.2 "Seven steps towards Object-based happiness"

Each of the following levels is taken verbatim from Object-oriented Software Construction (Meyer 1988).

2.3 Object-Oriented Terminology

The following sub-sections discuss the terms most relevant to this thesis.

2.3.1 Attributes or Instance Variables

An attribute is a field of an object, also called an instance variable in Smalltalk terminology. "Instance variable" is the most descriptive term: any object that exists at run-time must be an instance of a class, thus a variable contained inside an object is a variable of that instance or an "instance variable". Attributes can be made visible, external to an object. However, they should be available on a read-only basis, i.e. it should not be possible for an external object to change their state by either assignment or applying methods directly to them (a subject of discussion in section 5.2).

2.3.2 Operations or Methods

An operation or method is a series of actions that can modify the state (set of current attribute values) of, and/or request information about, an object.

2.3.3 Features

The features of an object are the sum of its attribute's states and its operations.

2.3.4 Visibility

An object is a mixture of internally and externally visible features. Declaring features as internally visible means that they are accessible only by that object. Declaring features as externally visible means that they are usable by any object that knows this object and can send messages to it.

2.3.5 Clients

A client is another object capable of sending messages to this object.

2.3.6 Messages

In an object-oriented system the only way for two objects to communicate is by using message passing. One object would send a request to another object by sending a message. If an object expects a response the called object, or some delegate, should send a message back. The enforced use of message passing with restricted visibility of some features gives rise to the possibility of building highly encapsulated objects, as then the only access one object has to another is by message passing.

2.3.7 Definition of an Object

The definition of an object used in this work is an amalgamation of the ideas presented above:

> An object is a run-time entity, with a set of attributes and methods collectively known as its features. An object can be modified only through the execution of its defined methods. Objects have internally and externally visible features. The object itself can use both its internal and externally visible features when performing a method. The client of an object may only access it through its externally visible features. All attributes are exported in read-only mode.[2]

This definition, whilst in the spirit of information hiding and encapsulation, is difficult to achieve and costly in compilation time as a high level of semantic analysis is necessary, particularly in achieving read-only attributes.

The definition above conflicts with some people's views of objects. Some of the literature, specifically Wirfs-Brock & Wilkerson (1989), suggest that an object's attributes should not be directly accessible to clients, but only accessed through explicit, programmer-defined operations. This limited approach can lead to unnecessary extra coding, if the programming language does not automatically generate a read operation for each attribute. On the other hand the features of some languages such as Smalltalk allow unrestricted access to the attributes, thus enabling programmers to manipulate the state as they wish.

It is necessary to avoid unrestricted access to attributes, as it compromises an object's encapsulation and impedes reusability (Wirfs-Brock & Wilkerson 1989). The implication of unrestricted access inside objects also affects the level of achievable concurrency. The level of concurrency decreases with increased object coupling and with the consequent increases in the amount of dependency between objects, discussed in section 5.2.2.

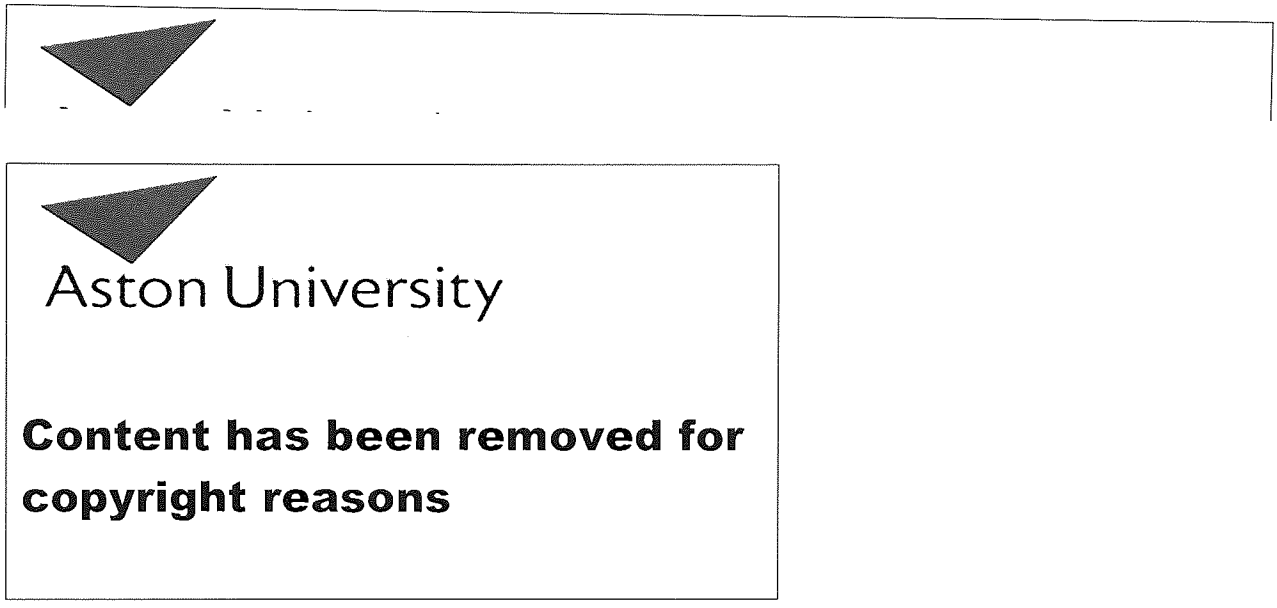
A balance can be struck between the opposing views of *no access* versus *unrestricted access,* a balance that does not compromise the integrity of an object's encapsulation: i.e.

[2]The idea of read-only attributes is presented in Meyer's (1988, pp 212) book, but even Eiffel, the language designed by Meyer, does not achieve this in actuality, see section 5.1.2.

restrict all attributes which have been declared as visible to read-only. However, this strategy is problematic in most programming languages when it becomes necessary to reflect the sharing of objects in a model, see section 5.2.1.

2.3.8 Classes

The three quotations below give a general impression of what the term "class" means.



As the quotations suggest, the most basic use for a class is as a template describing groups of related objects. A class is a description of the run-time behaviour of an object, which is an instance of that class. They are the "type-description" of objects, where "type" is the operations performable.

The description given by Blair et al. (1989) does not stress the idea put forward by Meyer (1988) that a class should be considered a static entity. In practice, different object-oriented languages use classes in a range of ways, from dynamic to totally static entities.

Smalltalk represents classes as dynamic entities with a behaviour at run-time. A Smalltalk class can contain class-methods, these can affect class-variables at run-time. Modifying, at run-time, the values held in a class's class-variables enables changes in behaviour in the instances of that class.

Eiffel is at the static end of the spectrum for representing classes. Eiffel classes are purely static entities. They describe groups of related objects, which can be instantiated at run-time. They do not contain either class-variables or class-methods. The lack of class variables does not weaken the static class languages as it is possible to simulate the effect of class variables by sharing an object, or using Eiffel's "once" feature, see Meyer's (1988, pp 309-314) book.

This thesis deals only with the view of classes as static entities.

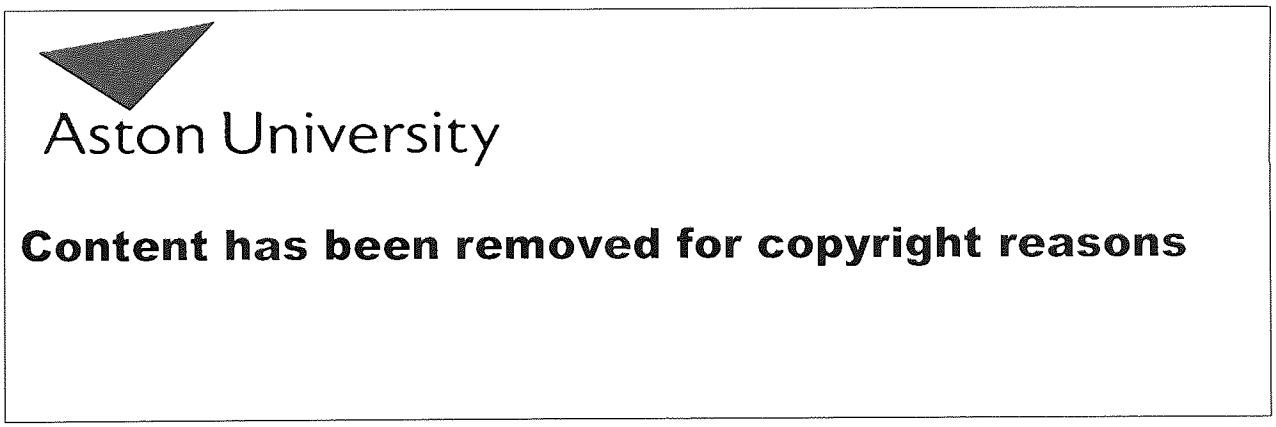
2.3.9 Genericity

Meyer's seven levels (section 2.2) mention genericity implicitly in level 6, as it is an implementation of parametric polymorphism (see 2.3.11). Genericity (in object-oriented programming languages) is usually associated with the module-level description of objects within programming languages. Its use is not purely an implementation consideration, it can greatly aid in the description of an object-oriented system and also in reusability.

There is some support for the explicit use of genericity in some object-oriented programming languages; e.g. Eiffel and C++. In practice, useful as the support for generic descriptions is, its support is a pure syntactic sugaring in any programming language that also supports multiple inheritance. The full descriptive power of genericity is a proper subset of that found in multiple inheritance. For a convincing argument on this topic refer to the book by B.Meyer (1988, pp 399-422).

2.3.10 Inheritance

The two quotations below by Micallef and Wegner outline the idea of inheritance.



Inheritance is a mechanism by which subclasses can be written which reuse, specialise, generalise, or form a subtype of a superclass. Snyder (1987) suggests that the four uses for an inheritance hierarchy are not totally complementary; indeed this clash of uses has led to some authors suggesting the limiting of its use to one or two of the four uses. In at least one approach (taken with the language Duo-Talk) multiple hierarchies enable use of both specialization and code reuse (Lunau 1989). Duo-Talk creates this separation by using one hierarchy for the interfaces to an object and a further one for the implementations of an object.

There are two forms of inheritance - single and multiple. Single inheritance is less flexible than multiple as it allows a class to inherit from only one immediate superclass. Multiple inheritance allows a class to inherit from multiple superclasses.

Multiple inheritance appears in most of the object-oriented languages due to its flexibility. For example, given a class that represents a "temporary-worker" and another class that represents a "secretary" a subclass "temporary-secretary" could be generated, quickly and easily. The "temporary-secretary" class then inherits its features from both of its superclasses.

There are difficulties in the use of inheritance:

- What happens when a class inherits from two classes with features of the same name (name clashes)?
- What happens if you wish to redefine some of an inherited class's features?
- How do you redefine a feature, but keep a copy of it for alternate use?

These problems are soluble; a simple solution lies in the ability to rename inherited features. If a name clash is to be avoided then rename the feature. If a feature is to be redefined then simply redefine it. If a feature is to be redefined and a copy kept for use, then firstly rename it (for use) and then redefine the original.

Micallef's definition earlier does not refer to the possibility of a class making visible features which are hidden in the superclass. This is convenient when a programmer needs to define a new visible interface, but can cause problems in the breakdown of a class's encapsulation.

Throughout this document I will use the more descriptive terms descendant (subclass) and ancestor (superclass). Two other useful terms are parents (the immediate superclasses of a class) and children (the immediate subclasses of a class).

2.3.11 Polymorphism

Polymorphism of a routine or module (class in object-oriented) is the ability for that routine or module to take on different forms. When a routine is polymorphic, it can handle different types of input. When a module is polymorphic, it can act as a holder for different types of data - e.g., a stack which holds integers, or records, or strings, etc.

There are various forms of Polymorphism.

Figure 2.2 shows a refinement of C. Strachy's categories of polymorphism as presented in Wegner's (1987) paper on object-oriented classification. Wegner breaks polymorphism into two main forms: Universal and Ad hoc.

Universal polymorphism can be recognised by the same routine being usable on different but semantically-related structures. Universal polymorphism can be further broken down into parametric (e.g., see ML) and inheritance polymorphism.

$$Polymorphism = \begin{cases} Universal & \begin{cases} Parametric \\ Inheritance \end{cases} \\ Adhoc & \begin{cases} Overloading \\ Coercion \end{cases} \end{cases}$$

Figure 2.2: Varieties of Polymorphism

Parametrically polymorphic routines, or modules, require the type of the parameters to be supplied as a parameter: for example, a routine that counts the number of elements in a list where the elements are all the same type. Supplying the parametric type to this counting routine enables the derivation of the actual type.

Inheritance polymorphism is the result of using an inheritance hierarchy. In the use of the inheritance hierarchy an algorithm may be written, for example, to search a sequenced structure. This search routine could be written at a very general level and held in the class representing sequences. Defining the necessary routines such as *Next-Element* in all the descendant classes of a sequenced structure specializes the search routine to work on any sequenced structures. Selecting the appropriate routine is then done at run-time using dynamic binding (see section 2.3.12), which searches the inheritance hierarchy for the specific routine to execute.

Ad hoc polymorphism can be recognised by the repeated use of a name to perform the same basic operation, on semantically-unrelated types. Ad hoc polymorphism is made up of two types - overloading and coercion.

A *Print* operation is an example of an overloaded operation. It might take various structures and produce visual representations. The structures supplied as a parameter might range from simple types to trees to even more complex entities. Printing then uses the appropriate procedure for the given type. Each type requires the writing of a different routine and the language system must select the appropriate one at either compile or run-time.

Coercion can occur when a routine requires a particular type of parameter and the supplied object's type is not of that type. To enable execution a supplied parameter's type is converted, by a suitable routine, into the required type and the converted value supplied to the routine.

An example of coercion can be seen in a routine that adds two real numbers: *add_two_reals*. Such a routine might expect two real numbers for its parameters. Supplying an integer and a real number to this routine results in the need to convert the integer into a real number.

This conversion could be done using a *float* routine, which given an integer returns the real number equivalent. At execution time, convert the integer to a real number, supply the resulting real number as a parameter and execute the *add_two_reals* routine.

Ad hoc polymorphism is also available in an inheritance hierarchy. A routine with the same name may be used often on semantically unrelated types. To choose the appropriate routine it is necessary to search through the inheritance hierarchy starting at the class of the current object. The search continues up the inheritance hierarchy until it finds the required routine, or reaches the top of the hierarchy, i.e., an undefined routine. Defining a print routine in all classes, or one of their ancestors, makes it possible to overload the print operation as described previously. It is therefore possible with inheritance to get the effect of both universal and ad hoc polymorphism.

2.3.12 Dynamic Binding

Dynamic binding is the ability to choose, at run-time, the most specific operation for application to an object. Consider the specialization that should occur in a well constructed inheritance hierarchy. A procedure written for a general structure - usually held at the top of the hierarchy - is typically less specific and less efficient than the same procedure written specifically for an object of a descendant class. For example, a search routine for a general ordered sequence would search through the sequence's elements one after another to find the required element. However a specific descendant class that holds extra information about the position of element's values could contain a very specific and much more efficient routine. These benefits come with dynamic binding.

Dynamic binding causes the selection of the best procedure available; this assumes that a specialized procedure is better than a general procedure. If there is no specialized routine then the general one is always there for use.

2.3.13 Cluster

A cluster is a set of conceptually related classes. B. Meyer first proposed the cluster (Meyer 1990, Meyer 1989*b*) as the basis of an object-oriented life-cycle. In a typical object-oriented language the biggest building block is the class and classes cannot be nested. When building systems it is beneficial to group related classes from particularly interesting subsystems; the name for such a group is a *cluster*. For example it may be useful to have a windowing cluster that contains all the classes used for the handling of windows on the local UNIX workstation. See (Henderson-Sellers & Edwards 1990) for a discussion of clusters and the idea of an object-oriented life-cycle.

2.4 Summary

An "ideal" object-oriented development system is a system that supports the use of clusters to organise related classes. A class is a modular implementation of an abstract data type. Classes can be generic. Each class defines attributes, and the methods that may be performed on the attributes. Features are the combination of attributes and methods contained in a class. It is possible to declare features to be externally visible; these features are then accessible by other objects in the system. Alternatively, declaring features to be private, or internal, limits access to them to the object itself. A class can inherit features from one or more ancestors; in turn it can choose to make those features visible to clients. When using inheritance a class may rename or redefine any of its ancestor's features.

An object is a run-time instance of a class. To request the performing of an object's methods or the supply of information, it is necessary to send a message to the object. Assignment to an object is possible by another object if the object's type (the one on the right of the assignment statement) is that of a descendant class - this is allowed through support for polymorphism. The particular operation applied to a polymorphic entity derives from the entity's type and, logically, a routine search through the inheritance hierarchy - this is dynamic binding.

Chapter 3

Parallelism Concepts

This chapter presents the ideas found in programming-language level support for parallelism. It does not, and neither does this work as a whole, look at hardware issues.

This section summarises the main techniques used in programming languages to achieve parallelism: the basic building block of the object-oriented paradigm is the object; the basic building block of parallelism is the process.

Parallelism is the execution of two or more processes at the same instant in time; concurrency is the *possibility* of executing two or more processes at the same instant in time.[1]

A Process is the sequential execution of a sequence of statements from a program.

Processes must communicate and synchronize during execution of a concurrent program, unless the processes are totally disjoint. Communication is necessary so that processes can exchange information and influence each other's actions. Synchronization enables the delaying of one process by another, enabling communication to take place or the performance of some coordinated action. There are three main issues underlying the design and expression of concurrent computations:

1. How are processes specified and created?
2. How is interprocess communication performed?
3. How is synchronization of processes obtained?

The rest of this chapter's discussion revolves around these three main issues: process specification, interprocess communication, and process synchronization.

[1] This distinction is convenient and I have attempted to adhere to it, however "concurrency" is used so extensively within the literature to mean the same as "parallelism" that there will probably be occurrences of such usage within this document.

3.1 Specifying Cooperating/Concurrent Processes

There are several notations in use which can denote concurrent execution (i.e., indicate the initiation or resumption of multiple processes): Co-routines with the *Resume* statement; *Fork* and *Join* statements; *Cobegin Coend* blocks; and Process declarations using syntactic-level structuring to indicate processes.

3.1.1 Co-routines

Co-routines are similar to subroutines as found in Fortran; they are sequential blocks of code. The analogy breaks down when considering the relationship between a caller and the called routine. Subroutines exhibit an asymmetric relationship with the caller; when a caller calls a subroutine, the subroutine executes from start to finish and returns to the caller. The relationship between co-routines, however, is symmetrical. Co-routines, when first called, start executing from the start of their code; they then transfer control explicitly to other co-routines as necessary. When subsequently called, a co-routine resumes from the point where it left off, and not at the start of its block of code.

Co-routines, then, indicate explicit transfer of control between cooperating sequential processes by executing a *Resume* statement - explicitly naming the other co-routine. Co-routines are designed to be used on single processor machines, and do not capture the idea of parallelism - allowing, as they do, the logical execution of only one process at a time. Each co-routine represents one process. (Example usage can be found in Simula-67 (Nygaard & Dahl 1978), Modula-2 (Wirth 1982) and in BLISS and SL5 (Andrews & Schneider 1983))

3.1.2 Fork and Join statements

The *Fork* statement initiates a new independent process; it can be used to initiate, dynamically, any number of processes. The *Join* statement causes the synchronization of one process with another, usually used when the forked process has completed and is ready to rejoin with the initiator.

The difficulty with this approach is in separating out the concurrently executable processes from the executable code, as *fork* and *join* are so tightly bound up in the code. (Used in PL/I; Mesa; and UNIX where *join* is implemented by the system call *wait*.)

3.1.3 Cobegin and Coend

Cobegin with its corresponding *Coend*, was introduced as a proposed extension to Algol-60 by Dijkstra (1965).[2] this was the earliest "structured" approach applied to concurrent process initiation.

The statements between a Cobegin Coend block are executed concurrently; the block finishes when all the enclosed statements are finished. Programs with this approach are in general much easier to follow than the equivalent using fork and join, thanks to the systematic method for denoting concurrency. The Cobegin Coend approach, however, is less powerful than the unrestricted use of fork and join; it lacks the ability to generate processes dynamically. It is, however, sufficient for capturing most forms of concurrency. (Used in variant forms in Communicating Sequential Processes (Hoare 1985, see CSP's concurrency operator ($a \parallel b$) and pp 225,226), Argus (Liskov & Scheifler 1983) and Edison (Brinch Hansen 1981*b*, Brinch Hansen 1981*a*).

3.1.4 Process Declarations

The processes in a system can be declared syntactically within a program's code in a similar way to that of a module or procedure, using a keyword such as Process or Task and the language's name scoping rules. Given this method of declaring a process there are three notable approaches to initiating processes. These three approaches have an impact on the number of processes at run-time:

1. A program has a fixed number of processes; these start simultaneously. The program terminates when all the processes have finished. Two examples of this approach, "Distributed Processes" (Brinch Hansen 1978) and "Synchronizing Resources" (Andrews & Schneider 1983), treat the program description as a single Cobegin-Coend block. The main disadvantage with this approach is the fixed number of processes, defined at compile time.

2. A program has a fixed number of processes, but it is possible to have multiple instances of any process. Two languages, Brinch Hansen's (1975) Concurrent Pascal and Wirth's (1977) Modula, achieve this mechanism by providing what amounts to a fork facility. This fork mechanism is only usable during program initialisation. The main disadvantage to this approach is the inability to create dynamically a completely new process at run-time.

[2] the terms in Dijkstra's (1965) paper were *parbegin* and *parend* respectively. However, *Cobegin* and *Coend* appear to be more extensively used in current literature.

3. A program declares the required process types and then creates instances of these processes whenever necessary during execution, thus enabling a dynamic number of processes (e.g. discussed in Wegner & Smolka's (1983) paper with respect to Ada) with a possible limitation on the number specified syntactically as in Occam (May 1983, May 1987).

3.1.5 Summary

The process declaration approach, in practice, is more elegant than Co-routines, Fork, or Cobegin. Programs written using this technique are easier to read and understand, particularly in their initiation and management of concurrency. A language designed to support the explicit description of concurrent solutions should at least have this syntactic approach to process description and creation.

The approach taken in this thesis avoids the explicit description and initiation of processes; if concurrency is possible it is automatically initiated. However, the deriving of processes makes use of the lexical boundaries contained in an object-oriented program to choose processes.

Further to the above approaches, threads are another useful concept for capturing cooperating/concurrent behaviour. Via various implementation and declaration strategies, they give rise to what can be termed "lightweight processes." They are processes in the sense of potentially multiple processes executing in parallel but differ in that a thread shares the same address space as the thread from which it was launched.

3.2 Shared Variables

This section looks at techniques of communication and synchronization based on shared variables.

Shared variables are data structures to which more than one process has access. Communication between processes can take place by each of the processes referencing these variables when necessary.

The access to a shared variable must be managed, specifically when multiple processes can access the variable simultaneously. There is a classic example contained in most of the literature; the incrementing of a variable by one.

If x is a shared variable and more than one process attempts to increment it simultaneously, the outcome is unpredictable. It is possible that two increments result in the addition of one to x, instead of two. This inappropriate behaviour is a result of the method of implementation of the increment operation: *load* reg_1*; add 1; store* reg_1. It is a result of the

interleaving of the lower-level instructions. To avoid this behaviour exclusive access to x must be ensured for the critical code, termed the critical region. Mutual Exclusion is the name given to the synchronization required to stop multiple critical regions executing simultaneously. One other type of synchronization is required in concurrent systems - managing operations that must be performed in a specific order. This type of synchronization is termed Condition Synchronization. An example of condition synchronization can be found in the typical producer-consumer problem. One process (the producer) produces data, storing it in a finite buffer; another process (the consumer) reads data from the buffer and processes it. This requires at least two conditions to be met: a producer cannot put data into the buffer if the buffer is full; a consumer cannot take data out of the buffer if the buffer is empty. Thus condition synchronization must manage these two conditions. With this problem it is also necessary to ensure mutual exclusion as well, so that the producer and consumer do not inadvertantly deposit and consume data simultaneously leading to possible corruptions.

The following section outlines the main techniques used for controlling access to shared variables, such that the above problems and ideas are dealt with.

3.2.1 Busy-Waiting

Synchronization between processes can be gained by the processes setting and testing flags. It is effective for implementing synchronization, but complex when trying to implement mutual exclusion. The implementation of condition synchronization is achieved by processes setting flags. To wait for a particular condition a process repeatedly tests the value of the associated flag until it is at the required setting - hence the term *busy-waiting* or *spinning*; these flags are also termed "spin locks".

Mutual exclusion is much more difficult to produce by the method of setting, clearing and testing flags. It is achievable using a combination of 3 variables and a strict entry and exit protocol (Andrews & Schneider 1983). The variables are a combination of 2 flags and a *turn* variable indicating which process can go into its critical region. However as stated by Andrews & Schneider (1983), synchronization protocols based upon a busy-waiting strategy are difficult to design, understand and prove correct. Also, protocols that use a busy-waiting strategy waste large quantities of processing time checking on the status of flags and variables.

3.2.2 Semaphores

Semaphores, first presented by Dijkstra (1968), were the first high-level mechanism for handling both condition synchronization and mutual exclusion. Semaphores consist of a counter and two primitives that can alter the counter's value. The two primitives are called P and V; the first letter in Dutch for *wait* and *signal* respectively. The semantics of these primitives

are as follows: *wait(s)* decrements the value of the semaphore s if it is greater than zero, otherwise it waits until s is greater than zero and then decrements it; *signal(s)* increments the value of the semaphore s by one. *Wait* and *signal* are usually implemented as indivisible operations in hardware or an operating system's kernel. They do not require busy-waiting loops and thus do not waste processor time as do the busy-waiting strategies.

These two primitives - *wait* and *signal* - are effective for ensuring mutual exclusion by initially setting the semaphore value to one and wrapping critical sections up in a *wait signal* pair. They are also effective in the handling of simple condition synchronization: set a semaphore, initially, to zero; the process needing the condition to be satisfied executes a wait; the process satisfying the condition then performs a signal on the condition semaphore.

More complex condition synchronization can be seen with the problem of a finite buffer, discussed in the introduction to this section on shared variables. Modelling of the condition synchronization is easy with two semaphores (Ben-Ari 1982), one per condition: a producer cannot put data into the buffer if the buffer is full - *some_space*; a consumer cannot take data out of the buffer if the buffer is empty - *some_data*. The mutual exclusion can be ensured by using a further semaphore, *exclusive_access*.

The flexibility possible in the use of semaphores is the basis for their major weakness: it is easy to write several processes that deadlock because the *wait* and *signals* are in the wrong order, or have been inadvertently omitted. Consequently, semaphores require careful use.

3.2.3 Conditional Critical Regions

The Conditional Critical Region, attributed separately to Hoare and Hansen (Andrews & Schneider 1983), overcomes the problem of erroneous ordering of semaphores by using a more structured notation.

Mutual exclusion is achieved by encapsulating variables that require exclusive access into regions (the variables are termed resources). The language system ensures the mutual exclusion of any resource by not simultaneously executing two regions that share a resource.

Condition Synchronization extends the simple critical region, adequate for mutual exclusion, to include a condition; for the critical-code to execute, the condition must be satisfied. The conditional critical region removes the inaccuracy of semaphore usage, putting the work upon the compiler implementor. Although the conditional critical region enables a higher level description of concurrent behaviour it is a technique that is expensive to execute: every time the critical code in any region finishes, all processes that are waiting to use the released resource(s) must re-evaluate their conditions. A further disadvantage is that the code controlling a resource can be scattered throughout a large program and does not need to be held lexically together; this increases the difficulty of comprehension.

B.Hansen implemented a version of the conditional critical regions in the programming language Edison (Andrews & Schneider 1983). This avoided the problem of the cost associated with re-evaluating conditions by placing each process on its own processor.

3.2.4 Monitors

The idea of monitors is credited to Brinch Hansen in 1973 within his book "Operating System Principles" (Brinch Hansen 1978). These ideas were later extended by Hoare (1974).

Monitors are a combination of local variables (which hold the state of the resource that the monitor is controlling) and several procedures that manipulate the variables and any resources under the monitor's control. A monitor holds the control code for any resource lexically together in a module; thus, the control of any resource should be easier to comprehend.

Externally, the monitor's variables are hidden; the only access to the monitor is through any visible monitor procedures. Mutual exclusion is ensured as only one monitor procedure can be executing at any instant. Condition synchronization is far less elegantly managed; the main synchronization technique involves using a combination of wait and signal with condition variables. Wait in this instance always blocks, causing the process to be suspended and queued and the exclusive access to the monitor to be relinquished. This leads to the scattering of control code through various procedures and risks inducing similar errors to those found in the use of semaphores for managing condition synchronization. There are a number of variants possible, a good recent reference is Bahr (1995).

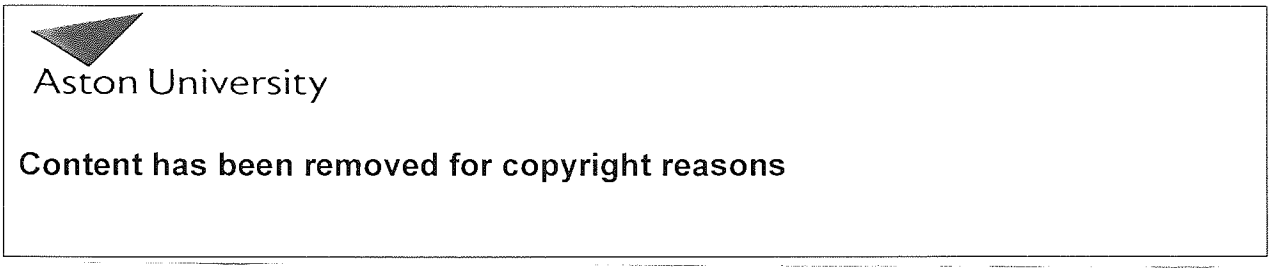
Monitors are a useful structuring idea; they are very effective in managing mutual exclusion and are effective but possibly error-prone in the management of condition synchronization. Example monitor-based languages include Concurrent Pascal and Modula.

3.2.5 Path Expressions

Path expressions extend the ability of monitors (the automatic handling of mutual exclusion) to incorporate condition synchronization. An expression contained in the body of a

concurrent module specifies the sequencing and execution information required to manage a concurrent system.

There are several techniques for writing path expressions, one of which is given in the quotation below.



The language compiler sorts out all internal synchronizations. This results in a flexible module description for managing resources suitable for concurrent access. When compared with monitors, the localisation of the condition synchronization information ensures that the access procedures are simpler and the whole writing technique is less prone to error.

3.3 Message passing

This section looks at message passing and techniques for communication and synchronization.

Message-passing systems usually use the single mechanism of message interchange for process communication and synchronization (SR includes more mechanisms (Andrews 1991, pp 577-578)). There are several possible variations in the semantics of any message-passing primitives; this has resulted in various approaches and techniques for handling concurrency.

Message-passing systems are built, with added variations, upon two basic primitives: *send* and *receive*. The *send* primitive transmits a message to another process. The *receive* primitive accepts a transmitted message from another process. Gentleman (1981) outlines three main issues which arise when designing message-passing primitives:

- How do processes specify the source and destination for a communication?
- How do processes synchronize using the communication primitives?
- How do processes structure the message?

The following sub-sections outline the ideas associated with these three points. A more detailed discussion of the possible approaches can be found in chapter 7 on the Concurrent Object Machine.

Message-passing primitives, when combined, provide flexible high level constructs, which makes programming easier. The most used construct is the Remote Procedure Call (see section 3.3.3).

3.3.1 Specifying the Source and Destination

There are three points to consider when deciding between different message-passing implementation techniques:

1. Is the process naming direct or indirect?
2. Is the naming relationship symmetric or anti-symmetric?
3. Are names static or dynamic?

The two approaches of process naming are as follows:

Direct naming: (Andrews & Schneider 1983) the explicit naming of the destination process in the *send* primitive's parameters, and explicit naming of the source in the *receive* primitive's parameters. Direct naming gives a one-to-one communication channelling. This method of identifying communicating processes is both easy to use and easy to implement. It enables flexibility in selecting when and with which other processes each process communicates.

Indirect naming: An intermediate structure forms a repository of messages - often called a mailbox. A *send* puts messages into an appropriate mailbox and *receive* gets messages out of the mailbox. Indirect naming thus enables a many-to-many form of communication. This method of communicating is more flexible than direct naming, but can result in poor control over the access to messages and imprecise synchronization between processes.

The second question of naming symmetry arises when considering problems found in client/server architectures. A server process will not necessarily be aware of the different client processes accessing it during its lifetime or the order in which clients expect to use it. The result is that an explicit statement of the expected source of a request, in a *receive,* is not possible. This results in the need for anti-symmetric naming in a message-passing model.

Symmetric naming: the identity of the source and destination of any message is explicitly contained in the parameters of the message-passing primitives.

Anti-symmetric naming: the *receive* primitive is allowed to receive from any process, (or mailbox). This solves the problem above in client-server architectures - a server using this approach does not necessarily need to know which processes are going to send it requests.

A further problem, worth considering, is exemplified by a client needing to use one of several available resources, e.g. a printer. If there are several equally functional printers, a client would want the printer that can handle the request first to print it. Initially this does not seem modellable using an explicit *send* with direct anti-symmetric naming - a client would need to send its request to each printer's queue, thus printing it several times. However, there is a solution that enables this one-to-many request and it is in the idea of a mailbox, discussed earlier. It is possible and relatively easy to construct a mailbox abstraction if a programming language supports direct anti-symmetric naming primitives and an effective abstraction mechanism. Thus, using direct anti-symmetric naming message-passing primitives, it is possible to obtain the effect of indirect (symmetric or anti-symmetric) message-passing primitives, obtaining the flexibility of the mailboxes and the preciseness of a direct naming system.

The third point to consider when choosing naming conventions is whether the names used in the primitives to indicate destinations and sources are static or dynamic. Static names are entirely specified at compile-time. This causes problems of inflexibility in executing the compiled program; channels of communication not known about at compile time are unusable at run-time. Dynamic naming uses names that are derived at run-time. Dynamic naming is in practice more difficult to implement than static naming but it results in a far more flexible system.

3.3.2 Synchronization of Processes

There are two main approaches to the synchronizing of processes using message-passing primitives, Asynchronous and Synchronous:

Asynchronous or non-blocking primitives are defined such that when executed they never cause the sender or receiver to wait. When a process uses a non-blocking *send* the message is sent to the receiver and the sending process carries on executing; this is irrespective of whether the message was buffered or even received. Equally with a non-blocking *receive*, a less useful primitive, a receiver performs a *receive*: if no message is present it continues executing; if a message is present it accepts it and then continues executing (e.g. PLITS).

Synchronous or blocking primitives cause a process which is performing a message-passing primitive to wait until the process with which it wishes to communicate reaches the point where it is performing a reciprocal primitive. When the two processes have reached their reciprocal primitives, i.e. the processes are synchronized, the message is exchanged. This point of synchronization is also called a rendezvous. (Example

languages: CSP, Occam and Ada).

Between the Asynchronous and Synchronous primitives are the buffered message-passing primitives (Andrews & Schneider 1983).

Buffered message-passing primitives associate a finite buffer with each process. This buffer fills up as messages are sent to it, causing a sender to become blocked until some of the information is removed. Buffered primitives, therefore, behave like asynchronous primitives until the buffer becomes full, at which point they behave synchronously and block.

Frequently, when presenting asynchronous and synchronous communication, the literature uses the idea of sending a letter and making a phone call respectively.

Consider the idea of sending a letter: a person sends a letter to another person and then continues with their life. The letter meanwhile is, hopefully, carried by the postal service towards its destination. The sender does not know whether the receiver has received the letter, unless the receiver sends a reply which is in turn received. The receiver gains information through the receipt of the letter, but the information, on receipt, is possibly out of date. This is like asynchronous communication.

Consider the second idea of making a phone call, which is like synchronous communication. A caller rings another individual (the receiver), and must wait until the receiver answers their phone before sending their message. When the receiver answers the two people (processes) are synchronized and the sender can now deliver the message. This synchronization is called a rendezvous. The telephone analogy can be taken further: if the sender makes a request of the receiver and the receiver attempts to answer the request, whilst keeping the caller waiting, then they are exhibiting an extended rendezvous or *Remote Procedure Call.*

3.3.3 Message Structure

The message format, often considered to be syntactic in nature, has implications on the efficiency of implementation, code readability and semantic implications in the way that the message-passing primitives are viewed and hence used (Gentleman 1981).

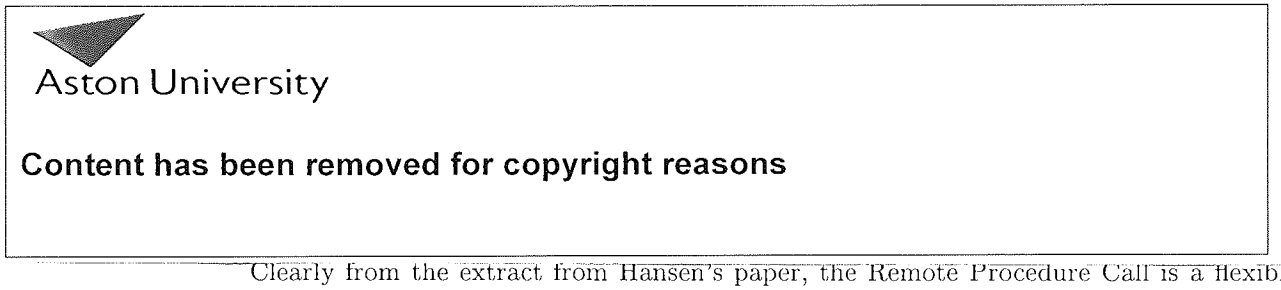
There are two possibilities the messages could exhibit: either fixed or variable formats.

Fixed format messages have several advantages: they are efficient at transferring messages (quickly); they are relatively easy to implement; and it is easy to ensure the atomicity of a communication. Their disadvantages include that they are inelegant when used to send messages larger in size than the fixed format, and they force implementation problems onto a user, e.g. how do I send longer messages?

Variable format messages also have several advantages: they are elegant in use, and single messages do not need to be broken up into multiple packets; consequently they are easier to understand and use. The disadvantages with using variable format messages include: they are harder to implement than the fixed format; they exhibit efficiency problems because of the varying message lengths; and ensuring atomicity of a message exchange is a much more complex task.

The message-passing primitives described in the previous sections are flexible enough to program any form of process interaction. However in a client/server architecture there is a noticeable pattern of interactions. A client will often send a request to a client and then wait for a result, and a server will often wait for a request, do some processing, and send a result. This pattern of interactions is known as a *remote procedure call.*

The idea of a remote procedure call is attributable to Brinch Hansen (1978), in his paper on "Distributed Processes": an extract from the paper's abstract is given below.



Clearly from the extract from Hansen's paper, the Remote Procedure Call is a flexible idea that has the semantic ability to capture and model many of the various ideas relating to concurrency and parallelism.

Syntactic-level support for RPC is provided by some modern programming languages (e.g., Ada, DP (Welsh & Lister 1981) and SR (Andrews 1991)); it forms a very effective syntactic-sugaring that aids programming and the construction of higher level abstractions.

3.3.4 Summary

The form of message passing that appears to be the most reliable and effective in the description of processes is a combination: using direct (symmetric and anti-symmetric) naming with synchronous communication and variable format message structures; this enables the simulation of any of the other ideas described in this section. It is not necessarily the most efficient approach, but is undoubtedly the most elegant.

Message passing tends to be a very expensive process in the amount of processor time used. The consequence of this is that, often, interprocess communication with message passing on a single machine uses semaphores and monitors, but when there is no shared memory proper message passing is used. This can be seen in the approach found in Xerox's work with ILU (Jansen, Severson & Spreitzer 1995).

Chapter 4

Object-oriented Handling of Parallelism

This chapter looks at current programming language based object-oriented solutions to the handling of parallelism.

There are two main approaches to the incorporation of parallelism into object-oriented languages: Actor or object-based models.

Actors, a calculus for describing concurrent activities, have been used as the basis for constructing object-oriented languages (e.g., Act 1 (Lieberman 1987), and ABCL/1 (Yonezawa, Briot & Shibayama 1986)). This seemed a very effective approach; however, after further study its applicability was felt to be too limited for this particular work. Actor systems work best on architectures with large numbers of processors exhibiting very low communication latency. Also the programming languages need ideally to tend towards the functional paradigm of description, as opposed to a state-based model. However, as discussed in section 4.1, some work has been done to distribute the Actor-based language ABCL/1 (Actor Based Concurrent Language).

The other main option is constructing and compiling object-oriented programming languages on top of object-based models defined originally for object-based languages. Current programming languages following this approach, whilst producing powerful programming languages for describing parallel systems, inevitably did not fulfil the thesis objectives: the languages make no effort to hide the parallelism, and this seems reasonable as it was not one of the objectives for the language designers. The object-based approach probably offers the best base for implementing a distributed concurrent object-oriented language. The main task then is to design a solid base model (see 1.1.2) which hides the concurrency but copes

with greater latency[1] in inter-processor communication.

4.1 Actor-based strategies

Actor as a term, according to Hewitt (1977), was first introduced in his thesis in the early 70's to describe the idea of a reasoning agent. Actors are based upon computation; they model objects as a function of incoming communication. This view of objects conflicts with the more traditional view of objects as a state-based combination of data structures and methods. Their behaviour is one of self-replacement in response to a message. They perform the appropriate action and generate a successor which is a modified copy of the object that received the asynchronous communication. They are very much computational units, being based around computation rather than data structures and associated function. High-level Actor languages may circumvent this problem, but the framework is not ideal for a concurrent state-based system with variable levels of granularity (Agha & Hewitt 1987).

Actors can model states (as indeed can pure functional programming languages) but as with functional languages the technique is awkward. So, although history-sensitivity can be incorporated into an Actor system, it is not an ideal framework for the description of any large concurrent state-based systems. Further state-based problems are found when a state must be shared between processes. Although state sharing is possible, Actors appear to be inefficient as regards implementation - state sharing requires a lot of copying and Actor replication. This problem arises because of the expected behaviour of Actors: when an Actor accepts a request and performs some action, it does not modify its state but "becomes" a new Actor. The new Actor is the result of the specified Actor behaviour for a given input. The old Actor is maintained.

Actor systems can dynamically distribute work, creating intermediate customers for the results of computations, thus making them seemingly ideal for implementation on parallel systems. However an Actor system is a combination of many very fine-grained processes. This means that a parallel system must be able to cope with a lot of context switches, necessary for the control of the executing threads; for example, computers that have tens of thousands of relatively small processors with low communication latency (Agha & Hewitt 1987). Having said this, some work has been done to implement ABCL/1 as a distributed language (Briot & Ratuld 1989).

Actor languages would appear to free the programmer from the worry of considering how the concurrency happens. This is not necessarily the case, though: the mapping of a program to its Actor image appears to be a one-way process (as mentioned earlier); it is not clear,

[1] Because of the target architecture, which is a network of UNIX workstations (see section 1.2.2).

when executing an Actor image of a program, which Actors are related to which program objects. This has an impact on debugging; the Actor "assembly language"-level description of concurrency may not reflect the mapping of objects and strict encapsulation in the high-level language. Indeed, whilst Actors maintain encapsulation of themselves, the encapsulation of the high level language could be compromised by the Actor structure. So, to debug an Actor-based program may require intricate knowledge of the concurrency implementation strategy.

The one-way mapping problem, above, is even more problematic when considering the process of distributing Actors across a system. In a typical network of processes, interprocessor communication overheads can be extremely costly. However, with the lack of mapping knowledge, distributing Actors and trying to produce a low level of interprocessor communication appears very difficult. As mentioned previously, this means Actors should be executed on networks of processors with very low levels of communication latency.

To summarise, whilst Actors seem an excellent model of concurrency for certain types of object-oriented language, they do not form a good base for implementing programming languages that use large state-based models. The ideal architecture for executing Actor systems should have many processors with fast context switching and a low interprocessor communication latency. The Actors themselves must be inherently small or the communication overheads will become overwhelming.

Consequently, Actors are not ideal when constructing software for networks of computers where context switching and interprocess communication is (comparatively) slow. In this context large grained parallelism and heavier weight processes[2] are better. Therefore, Actors do not offer a solution to the implementation of more typical object-oriented languages, in a concurrent format, executable on networks of workstations or processors with a high communication latency.

4.2 Object-based strategies

Object-based concurrent programming combines both concurrent and object-based programming approaches into one paradigm. It includes the ideas of encapsulation, classes and inheritance (in object-oriented languages) with the ideas of threads, synchronization, and communication (Peter Wegner 1990).

The run-time support for an object-based programming system needs to provide for

- Object management;

[2]It should be remembered that the target architecture is a distributed computer formed from a network of UNIX workstations (see section 1.2.2).

- Object interaction management;
- Resource management.

Distributed object-based programming systems as suggested by R.S. Chin and Samuel T.Chanson (1991), exhibit all the characteristics of an object-based programming system as well as supporting a distributed computing environment. Example distributed object-based programming systems include Amoeba (Tanenbaum & Mullender 1989/90), Clouds (Dasgupta 1986), CHORUS (Campbell, Russo & Johnston 1987), Emerald (Black, Hutchinson, Levy & Carter 1987). Typically, distributed object-based programming systems have the following characteristics: [3]

- A technique for distributing objects;
- A transparency of location so that programs can be made more general;
- Maintenance of data integrity, such that a persistent object is always in a valid state before it performs an action;
- Fault tolerance;
- Maximised availability of objects irrespective of a node failure;
- Recoverability of an object state if a node fails;
- Object autonomy - the ability for an object to state which objects can be clients;
- Program Concurrency - multiple objects should be assignable to multiple processors for concurrent execution;
- Object Concurrency in that an object should be able to service more than one request at a time;
- Improved performance in that a program, if well designed, should perform better on a distributed object-based programming system than on a conventional system.

Some of the object-based solutions (e.g. Emerald) are close to providing an object-oriented solution. However, Emerald does not support inheritance in the object-oriented sense. Instead Emerald supports a type hierarchy and a compositional model of construction to enable sharing. The compositional model facilitates the construction of objects from components, enabling encapsulation of the items as a single entity.

[3] For more detail relating to this list, which is a paraphrase of the original, see the original paper by Chin & Chanson (1991).

In practice, all the object-based solutions looked at put the concurrency control into the hands of the programmer. Implementing the automatic handling of concurrency uses some of the ideas and technology of the distributed object-based programming system paradigms; however, the technology is hidden below the level that a programmer normally deals with.

An effective approach for the base of any large object-based system has now become available from Xerox Parc, termed ILU (Jansen et al. 1995). This enables packaging of objects and deals with inter-object communication, performing optimisations for local machine interactions. However, it was unavailable as a core tool at the start of this work and thus is not used for implementation of the basic execution framework (see chapter 7).

4.3 Object-oriented Languages with Concurrency

There are several object-oriented languages, other than the Actor-based languages discussed earlier, that support concurrency. They (mostly) use a process declaration approach to introducing processes. Consequently, all of these languages force any concurrency control into the hands of the programmer, thus making a possibly difficult development job even more complex.

Solutions include direct extensions of "standard" languages - i.e. classes built to incorporate process management primitives; this has been done with C++ (e.g.Ishikawa, Tokuda & Mercer's (90) work on RTC++, an extension of C++ supporting real-time active entities) and can be seen on top of Eiffel (Magnusson 1990, Caromel 1989, Caromel 1990). Smalltalk too has had its share of work to extend it for concurrent work in for example ConcurrentSmalltalk (Yokote & Tokoro 1987). The solutions are effective for concurrency but do not address automatic concurrency management.

Other solutions to the handling of concurrency include new purpose-designed languages that require novel (to object-oriented languages) syntactic extensions in the language e.g. Hybrid (Nierstrasz 1987). Hybrid brings together a mixture of all the possible constructs that one could imagine for handling concurrency and thus provides a multitude of different ways to tackle problems.

One of the best solutions in this category of purpose-designed programming languages is America & van der Linden's (1990) programming language POOL (Parallel Object-Oriented Language). As a language it has been characterised as a simple version of Ada (Eliens 1994). It supports the notion of active objects (Eliens 1994); active objects have their own thread of control in parallel with threads in other active objects. POOL objects handle messages explicitly by interrupting their activity using an *accept* statement, as in Ada. The real strength in POOL is the extensive work done in laying a strong theoretical foundation

describing the semantics for a parallel object-oriented computing (America & Rutten 1990).

With respect to this work, the two major weaknesses with the POOL model are that it allows for only one thread inside an object thus limiting the potential level of parallelism, and the handling of parallelism is in the hands of the programmer and is not automatic.

Active objects have also led to the ability to distribute the objects across different machines, giving rise to another extension to Smalltalk called Distributed Smalltalk with its *proxy* objects that deal with communications between objects on different machines (Steel 1991). These *proxy* objects are similar in nature to the *couriers* in the model presented in chapter 7.

Further solutions include DRAGOON (Atkinson, Goldsack, Di Maio & Banyan 1991), an object-oriented extension to Ada; Ada 95, the new standard for Ada (*Ada 95 Reference Manual* n.d.); and COOL (Lea & Weightman 91), a kernel support for object-oriented environments (based on CHORUS (Zimmermann, Banino, Caristan, Guillemont & Morisset 1981)). There are other solutions including languages such as CLOS - a form of lisp; Orient-K - a knowledge-based language etc, but none of them address the problem of enabling concurrency without forcing the programmer to control it. All of the solutions in this section rely on the skill of the programmer for the tackling and handling of parallelism.

Chapter 5

Thesis Objective

The objective of this thesis as outlined in chapter 1 is to provide a self-contained, adequate theoretical object-oriented model for concurrency such that a model for implementation of a compiler for an object-oriented programming language can be produced, without extensions to the syntax or semantics of the language. Thus it will be shown that it is possible to automatically translate a sequential program into a parallelised program.

The theoretical model of concurrency will be called a "Concurrent Object Machine" (COM). The model is not expected to be optimal in either speed or space usage, but is expected to provide a good target for the model of a parallelising compiler for an object-oriented programming language, making the construction of such a compiler practical. The model for the compiler's implementation will, as with the COM-model, be based upon an object-oriented view of the problem. It will utilise the inherent logical structure and message passing found in object-oriented programs. This use of object-oriented ideas will enable an elegant and practical approach to the automatic parallelisation of a useful subset of object-oriented programs. These two models will be built using the Eiffel programming language and will use a subset of Eiffel as the language to be implemented. The generated code will be written to utilise a network of UNIX workstations as a parallel computer.

The rest of this chapter looks at the ideas to be implemented; it looks at the Eiffel programs that such a compiler should deal with including highlighting an apparent weakness in the current implementation of Eiffel compilers with respect to definitions given by Meyer (1988); it discusses the principles which, when applied to object-oriented programs, improve the effectiveness of such a compiler. Consideration is given to the view of executing object-oriented programs and how an alternate execution-model view from the "normal-execution" model in Eiffel has been adopted. Finally a brief overview is given to the approach applied within this thesis including a description of the Eiffel subset which will be dealt with.

5.1 The Starting Point: An Eiffel Program

Translating from Eiffel into a parallelised form starts with an Eiffel program. Eiffel programs are described using classes. There is one class - the *root-class* - which forms the glue that joins a collection of classes together into a program. This class has a *main* routine called **Create**[1], called to effect the start of the program, as **main** would typically be called to start a C program.

Eiffel classes (see chapter 2) contain definitions of attributes and methods. The attributes may be of a simple type - INTEGER, CHARACTER, REAL or BOOLEAN; or a type described by an Eiffel class (even a class of which it is a member, i.e. recursive definitions which can be either directly or indirectly mutually recursive). Methods can be categorised as one of two types: those that return a value and those that do not, termed queries and commands respectively. A query can return a value which is either a simple type or a type described by a class.[2] A class can be related to one or more parent classes by inheritance; the inheritance however cannot be recursive.

The recursive relationships, outlined in figure 5.1, lead to an interesting algorithmic problem which is discussed in section 10.1.3. The problem arises because of the recursiveness in the class relationships and the necessity for the Eiffel compiler to derive all class relationships and decide which classes must be (re-)compiled.

5.1.1 Good Eiffel Program ≡ Potential Parallelism

To achieve programs that actually execute in parallel, instead of as a collection of "sub-routines", it is necessary to write reasonably good Eiffel. This measure of "goodness", in the form of programming principles, is presented in section 5.2; the principles should be adhered to if maximum parallelism is desired. If a programmer chooses not to adhere to the principles then the level of potential parallelism will be reduced.

5.1.2 A Semantic flaw in the implementation of Eiffel?

It is stated by Meyer (1988), the originator of Eiffel, that:

> '...exporting an attribute entitles clients to access its value (in read mode), but not to modify (write onto) it.' (Meyer 1988, pp 212)

[1] The restriction of calling the main routine **Create** has been removed in Eiffel v3.

[2] NOTE: The view presented is simplified to bring out the most salient features of an object-oriented programming language. For example Eiffel supports a concept called "expanded types", which are flattened classes, i.e. they can be treated as one might expect to treat simple-types. Indeed STRINGs, for the purpose of this work, are treated as simple-type entities.

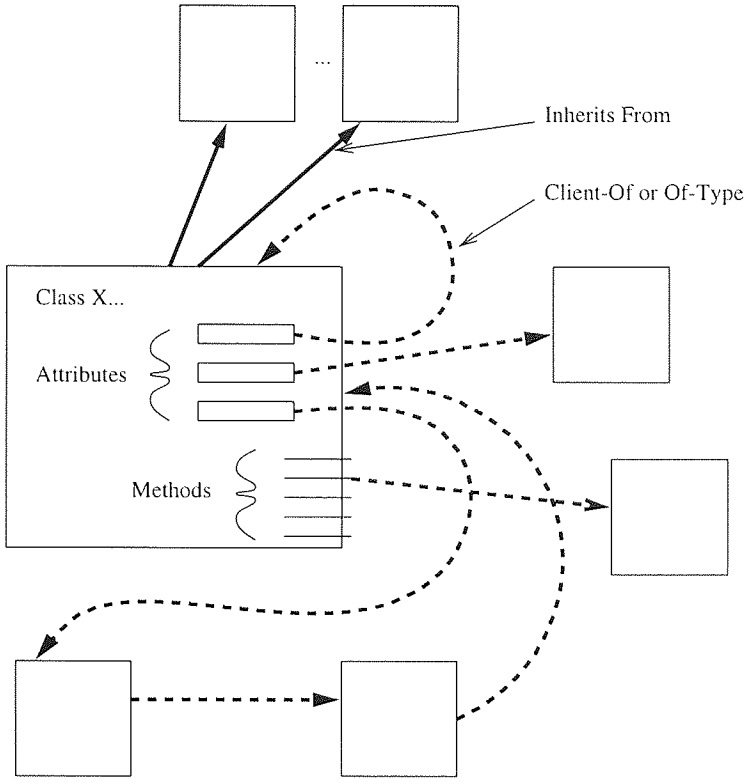


Figure 5.1: Class relationships in Eiffel

This is not the case with standard Eiffel implementations: whilst all objects exported cannot be assigned to, the read-only mode is easily broken if an exported attribute is a "non-simple" type - any method exported by the class describing the exported object can be applied to that object, including state change methods.

Figure 5.2 epitomises this apparent flaw: the object b in class A which is exported can be altered by the following statement `a.b.c.d.change_d_method(...);` where a is of type class A, thus allowing an external program to access not only inside instances of class A, for manipulative purposes - compromising encapsulation - but also down multiple layers. Thus a programmer using class A can make assumptions about how the program has been written, causing a loss of information hiding and a possible loss of continuity (i.e. the idea that small changes in specification should lead to correspondingly small changes within an implementation (Meyer 1988)).

It is possible to avoid such poor programming practice by choice and self-imposed discipline; even better, Eiffel has sufficiently well-defined semantics such that semantic analysis can highlight such errant programming style, though obviously at the expense of longer compilation times.

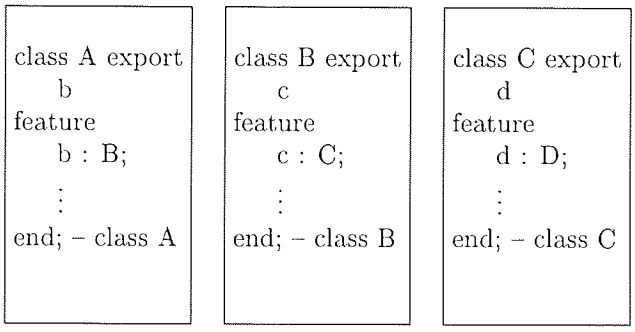


Figure 5.2: Weakly-encapsulated attributes

5.2 Programming Principles

The principles discussed are presented from two points of view: sound programming practice, so that it can be demonstrated that the assumptions made within this thesis do not lead to strange techniques for software development; and how such principles increase the potential parallelism. The principles all relate to accessing the interior of objects, and consequently to the style of programming used in classes, particularly the interface provided to instances.

5.2.1 Sharing: Multiple objects utilising a shared object

Modelling the sharing of objects is something that is necessary for both modelling the real-world, and also (from the point of view of understandability) when reflecting an abstract concept (e.g. a linked list) within a model. However, most object-oriented languages do not have a syntax or semantics to support the clear modelling of sharing; instead, sharing semantics must be contrived by the use of some feature within the programming language - typically exported references within objects.

For example, consider two objects of type PERSON and an object of type TELEVISION, both of whom wish to be able to utilise this TELEVISION. Somehow in an object-oriented system it is required that the potential interaction can be modelled. The two people might be expected to interact with the television, TV for short. From the perspective of information hiding it seems to be a poor design decision, particularly given section 5.1.2, to put an attribute of type TELEVISION inside an Eiffel class PERSON - people do not generally contain TVs. However, the most convenient way to model sharing is to incorporate in the class PERSON a reference to a TV, by defining an attribute of that type. This attribute can then be assigned to by a "setting-operation", which PERSON exports. The creation of the TV object should be external to the people, possibly in some greater class which brings together the idea of these people and TVs.

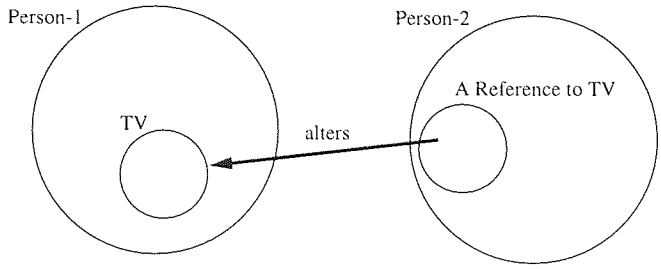


Figure 5.3: Poor object sharing

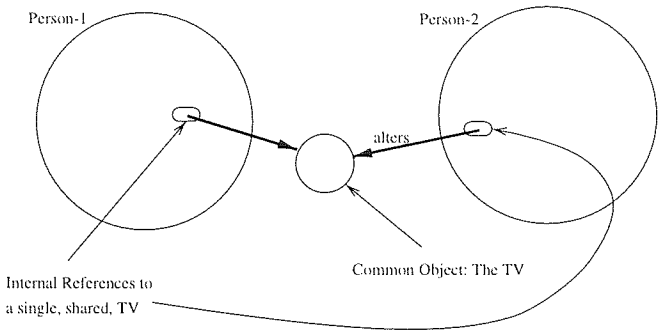


Figure 5.4: Better object sharing

To model a shared object in an object-oriented programming language such as Eiffel, due to the lack of a good syntactic-sugaring with suitably defined semantics to reflect the concept of sharing, it is necessary to nominally break encapsulation, and to write a class which seems to be "nonsense" from a modelling point of view, e.g. PERSON containing a TV. Indeed, because of the need for sharing, the semantic-flaw of section 5.1.2 becomes a virtue. The way to perceive this type of class is not as figure 5.3 but as 5.4, even though the code reflects figure 5.3 (section 5.3 looks in more detail at the perception of execution in the execution models).

Consider, however, a teletext decoder defined within the TELEVISION class: it is an object that is contained within another object, the TV. It would not be expected that a PERSON would access the decoder, instead they would use it by manipulating the TV's defined interface. Therefore this item should not be exported, even though it would give a quick convenient coding technique.

This use of encapsulation, called *containment* from now on (if the encapsulated object is externally inaccessible), is the fundamental structuring mechanism that, along with communication via message passing only, enables the automatic generation of parallelism.

5.2.2 Containment

The containment of an object within another object should be a central theme when building large object-oriented software systems. Support for containment, meaning the ability to strongly encapsulate a contained object, helps not only to maximise parallelism but also to maximise the achievement of the criteria for design put forward by Meyer (1988, chapter 2) - modular decomposability, composability, understandability, continuity, protection - and four of the five corresponding principles - few, small, explicit interfaces with maximised information hiding.

5.2.3 Ideal: Accessor/Method division

Meyer (1988) discusses a query/command division: a query, as might be suggested by the term, is a method that returns information about the object receiving the request; a command causes a change of state in the receiving object. A query, it is suggested by Meyer, should be restricted such that it does not cause a state change within the object that receives the request.[3]

This division between queries and commands is not rigorously enforced in object-oriented programming languages. However, given the level of semantic analysis possible it could be done in Eiffel. This would be beneficial, particularly in the context of this work, as it ensures maximum parallelism because of the reduced side-effects.

5.2.4 Summary

To summarize, if a programmer attempts to maximise fulfilment of the criteria put forward in chapter 2 in Meyer's (1988) book, then they inherently maximise the potential for parallelism. Certain aspects are not emphasized in the book, but as argued are relevant to Meyer's own criteria: ensuring containment of objects (minimising inter-relationships thus increasing potential parallelism; exporting objects only if they are shared objects) which also minimises inter-relationships; and enforcing a query/command division - giving the potential for reader/writer divisions in operations such that they can be run in parallel.

[3]This is a slight oversimplification, as Meyer (1988, pp 135-139) suggests that state change is acceptable in the concrete state as long as the abstract state is not altered. The distinction between abstract and concrete state appears artificial and unnecessary; no example is given beyond that of complex numbers - which could easily have been implemented using an alternative strategy without the increase in complexity. There is however a possible efficiency question, e.g. wasteful conversions inside queries, between polar and cartesian representations of complex numbers, which are not reflected in the object's abstract state even if the next method call would also require this translation - complex in polar form; request x means conversion to cartesian form; next operation if a command to *add* would also require to do this conversion.

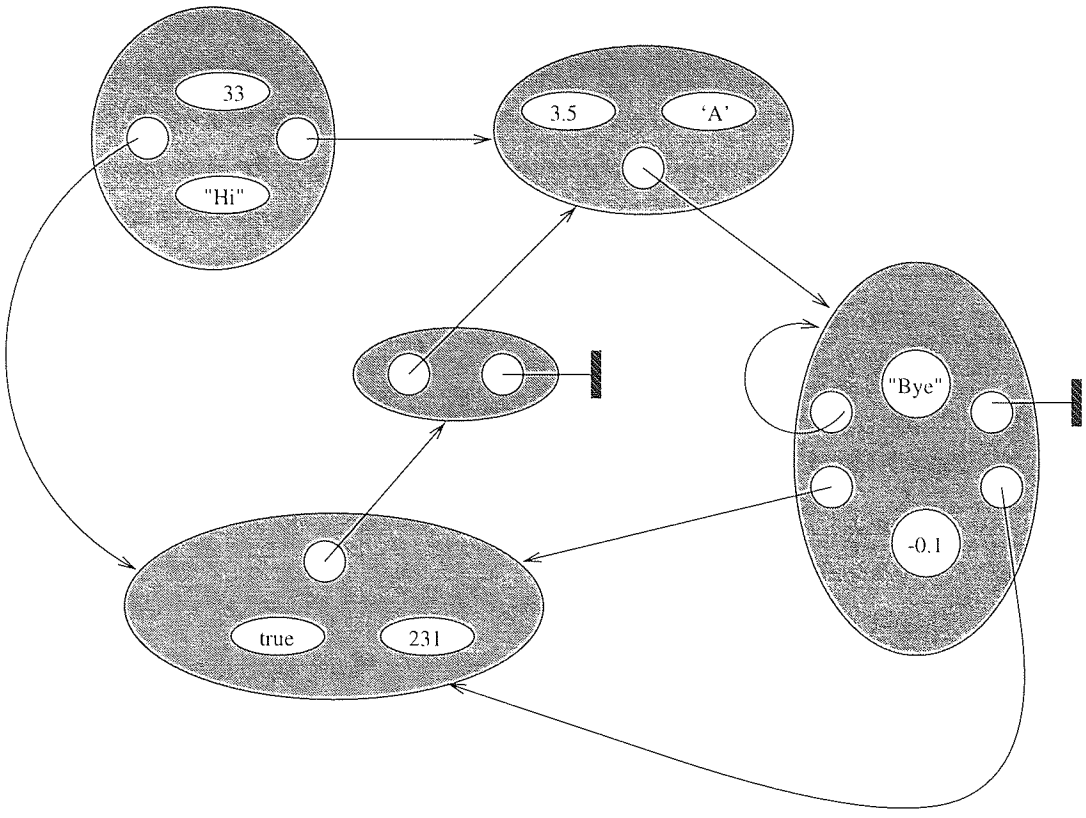


Figure 5.5: "Normal-model" view

5.3 The Execution Model

The "normal-execution" model (Meyer 1988) is an inadequate base-model for this work on parallelism. It does not clearly model the concept of encapsulation, which is central to this work.

As discussed briefly in section 1.1.2 and in more detail by Brunskill *et al.* (1995) the basic model's effectiveness is important if reasoning, refinement and implementation are to be effective. The "normal-model" view is presented in the following section, with an explanation of why it is problematic; this is followed by a section on an alternate view that better reflects aspects of encapsulation and information hiding and enables world-views to be mapped with greater clarity onto an execution-model.

5.3.1 The "Normal-Execution" Model

Figure 5.5 presents the "normal-execution" model view.[4] It depicts five arbitrary objects and inter- and intra-relationships between those objects by using references. Although it is

[4]The diagram is semantically equivalent to that found on page 70 of Meyer's (1988) book, but for convenience of generation the ellipses, circles and lines vary in size and placement.

an inference based on the diagram and reading this book, the model does not reflect the idea of containment. Simple objects, here including STRINGs, are represented as being contained within an object, but entities described by a class are drawn outside of the relevant ellipse; hence it is not clear whether an object is conceptually contained within an object, or is shared between a number of objects.

This means that one of the tenets[5] of object-oriented software development, encapsulation, is missing from the execution model. This absence can lead to a "lazy" programming style that does not maintain containment of entities, with strong encapsulation, but instead enables programmers to ignore the concepts of information hiding (see section 5.2.2). In short it can lead to the criteria and principles outlined in chapter 2 of Meyer's book being broken, and consequently to poorer quality software systems (see also section 5.2.2).

5.3.2 An Alternate Model View

An alternative execution-model view to that outlined in section 5.3.1 might attempt to incorporate any needed features that the "normal" view does not possess, but it must not be a semantically weaker model, otherwise it may become a hindrance in the implementation of Eiffel. The alteration made is to explicitly reflect the idea of containment and sharing in the execution model.

To aid parallelisation, when considering Eiffel programming style and classes, only export an object from an Eiffel class if it is supposed to be shared, especially if the item is "non-simple". This is necessary (see section 5.2.2) as any exported entity, whilst semantically read-only, is in practice, when "non-simple", alterable using any routines that are exported from the contained object's class description. If an entity is not exported then it is viewed as being an object fully contained within an instance of that class. This alteration to the execution-model view, if carried through to programming and semantically to implementation, will maximise information hiding and increase the potential for parallelisation.

5.3.3 The Concurrent Object Machine

The alternate-execution-model view used in this thesis will form the basis for the abstract machine onto which the parallelised Eiffel programs are to be mapped. The machine is termed a Concurrent Object Machine, or COM for short (see chapter 7). This abstract machine brings together Gentleman's (1981) ideas of workers and communication, with a view of

[5]The tenet that encapsulation is important is not universally held in the diverse object-oriented community. It is however beneficial if encapsulation is enforced (Meyer 1988, chapter 2) when writing large high-quality object-oriented systems.

object-orientation that makes enforced encapsulation a necessary virtue, and incorporates the ideas of client-server architectures from operating systems.

5.4 Overview of Approach

As mentioned at the start of this chapter the programming language to be implemented is a subset of Eiffel. The architecture on which the implementation is to be executed is that of twenty VAXstation 3100s running UNIX and connected with TCP/IP over ethernet. The implementation language is Eiffel v2.2.

5.4.1 Language to be implemented: Eiffel

Eiffel was chosen as a suitable programming language to implement because it has reasonably well-defined semantics (Meyer 1989*a*, Meyer 1992). The particular version of Eiffel looked at is 2.2, as version 3 was not available for most of the period of this work. Although the work is focused on version 2.2 Eiffel, in practice the ideas should translate with little change to version 3, as the majority of alterations between the versions are syntactic.

The implementation of Eiffel is a non-trival task when compared with a lot of programming languages partly due to its size, its features, and those aspects incorporated to aid in building large systems (e.g. unlike C systems no make file is required). This has led to a subset being chosen for investigation. Those features of Eiffel not implemented include parametric polymorphism or genericity; assertions and the associated pre- and post-conditions, invariants and loop invariants; low level manipulations such as interfacing for C; and once routines. Consequently the features found in most object-oriented programming languages have been considered and investigated, particularly objects, attributes, methods, message passing, classes and inheritance.

5.4.2 Implementation Architecture

The architecture on which the implementation was done is that of a set of diskless "VAXstation 3100" workstations, with 25 megabytes of RAM, running UNIX. The workstations are connected by an ethernet backbone and supported by a common server which holds the file-system.

The advantages of this are that with the memory not being common the ideas presented will map onto shared memory multi-processor machines as well as non-shared memory multi-processor machines.

A major disadvantage of this strategy is that true empirical measurements of effectiveness are not really possible. These workstations are on a backbone common to many computers

and thus have to deal with a great deal of background "noise" as well as any communications that might be necessary to a parallelised program.

5.4.3 Host Operating System: UNIX

The operating system used both for implementing and for the execution of an implementation is UNIX. The main features that have been used are Berkeley sockets, the fork call and processes. This means that it should be possible, given a suitable compiler, to map the ideas onto any operating system supporting these features. As will be discussed, this is not an ideal set of features but should include: threads, unsupported at time of development on this version of UNIX; fast interprocess communications such as system V message passing, or shared memory for fast local machine inter-object communications. Another ideal feature unavailable on this version of UNIX would be a means to automatically distribute and load balance threads and/or processes - an area of research at present (e.g. Blumfore, Joerg, Kuszmaul, Leirerson, Randall & Zhou's (1995) multi-threaded parallel programming on Connection Machine CM5 and other machines).[6]

5.4.4 Implementation Language: Eiffel

Eiffel is the implementation vehicle for the thesis along with C as required. Eiffel was chosen because the standard Eiffel libraries provide extensive tools for creating parsers, it has the ability to interface with C for low-level manipulations, and it supports a useful level of abstraction when implementing solutions with its support for object-orientation.

5.4.5 Program Analysis

Parsing

The parsing is very much as for standard compilers. The basic technique is similar to the effect achieved with recursive descent, with the associated building of symbol tables and abstract syntax trees (AST) which are annotated with the source program's information.

Semantic Analysis

This involves the usual ideas of analysing code to build up a picture of the algorithm involved, particularly to help in optimisation. However, it also incorporates what is basically a process of taking the transitive closure of all possible paths and relations to work out as much information as possible to facilitate transformation into a parallelised program. For example,

[6]The research nature of distribution and load-balancing algorithms makes their study in this area a major research in its own right and means that it is necessary to leave this to future work.

it checks if an object supplies a method to fulfil a request, or whether the application of a number of methods causes a state change.

Code Generation

This uses the tables and ASTs built in the previous phases of parsing and semantic analysis and generates a class to represent each method and each class, as required by the execution model. The classes become descendants of classes which provide inter- and intra-object communication, and form the basis for the abstract machine onto which the parallel model is mapped.

5.4.6 User's View

Compilation: Eiffel code to Parallel programs

When the system has been built, a user who would normally type *es* to compile a complete Eiffel system will now type *build_all* and everything will happen automatically from parsing through to final compilation of the code-generated Eiffel.

Execution of the Parallelised Eiffel Program

The execution method, as with the compilation strategy, will be straightforward. A user, as if running a normally-compiled Eiffel program, will type the name of the root class and the system will automatically execute associated programs on local or remote machines, as required, to effect the creation of objects and the execution of the original algorithm.

Part II

Design

Introduction to part II

This part looks at the design of a solution to deal with translating a sequential object-oriented program in Eiffel into a parallelised program. Chapter 6 looks at the design of the translator with chapter 7 presenting the design of the abstract machine onto which the translation maps. Chapter 8, looks at issues such as deadlock. It suggests that it is possible to remove the problem by a priori design decisions within the code-generation of the compiler. The final chapter in this part, chapter 9, looks at the code-generation phase of the compiler, detailing how some of an object-oriented programming language's constructs would be translated.

The work in this and subsequent parts is that of the author. Chapter 7 as it suggests is influenced by (Gentleman 1981, Burkowski, Cormack & Dueck 1989) but the idea of a COM is mine. The work in chapter 8 benefited from useful discussions and work with Prof. Colin J. Theaker but again is my own work.

Chapter 6

The Translator

This chapter examines the design of a translator, which as with all of the suggested approaches within this work is object-oriented, and its necessary features such that sufficient semantic information can be produced, via semantic analysis, to enable code generation of a parallelised form of the input program.

6.1 The Translator: an overview

The process to be used is typical of that found in most, if not all, translators; it passes through various phases from parsing to code generation (see figure 6.1). The expected input, for this translator, is an Eiffel program[1] combined with the standard description file (see section 6.1.1). The generated code will be, for this work, a parallelised form of Eiffel, where each class is rewritten in the process of translation as multiple classes, made to inherit pre-written classes which provide run-time support for parallelism.

The objective of the parsing and semantic analysis stages is to produce sufficient semantic information to support the automatic construction of a parallelised version of the input program.

The translator is object-based in its structure. The organisation of this structure is based upon the structure of an Eiffel program:

> An Eiffel program is a collection of classes; one of the classes is termed the root-class, its creation routine[2] starts and controls the program's processing. An Eiffel class is a composite of feature definitions, both attributes and methods, and various clauses describing relationships with other classes and the external

[1] The program is limited to a subset of Eiffel, as mentioned in section 5.4.1: i.e. no genericity, assertions, low-level manipulations, or once routines.

[2] Version 2 Eiffel classes define only a single creation method; version 3 classes can have multiple.

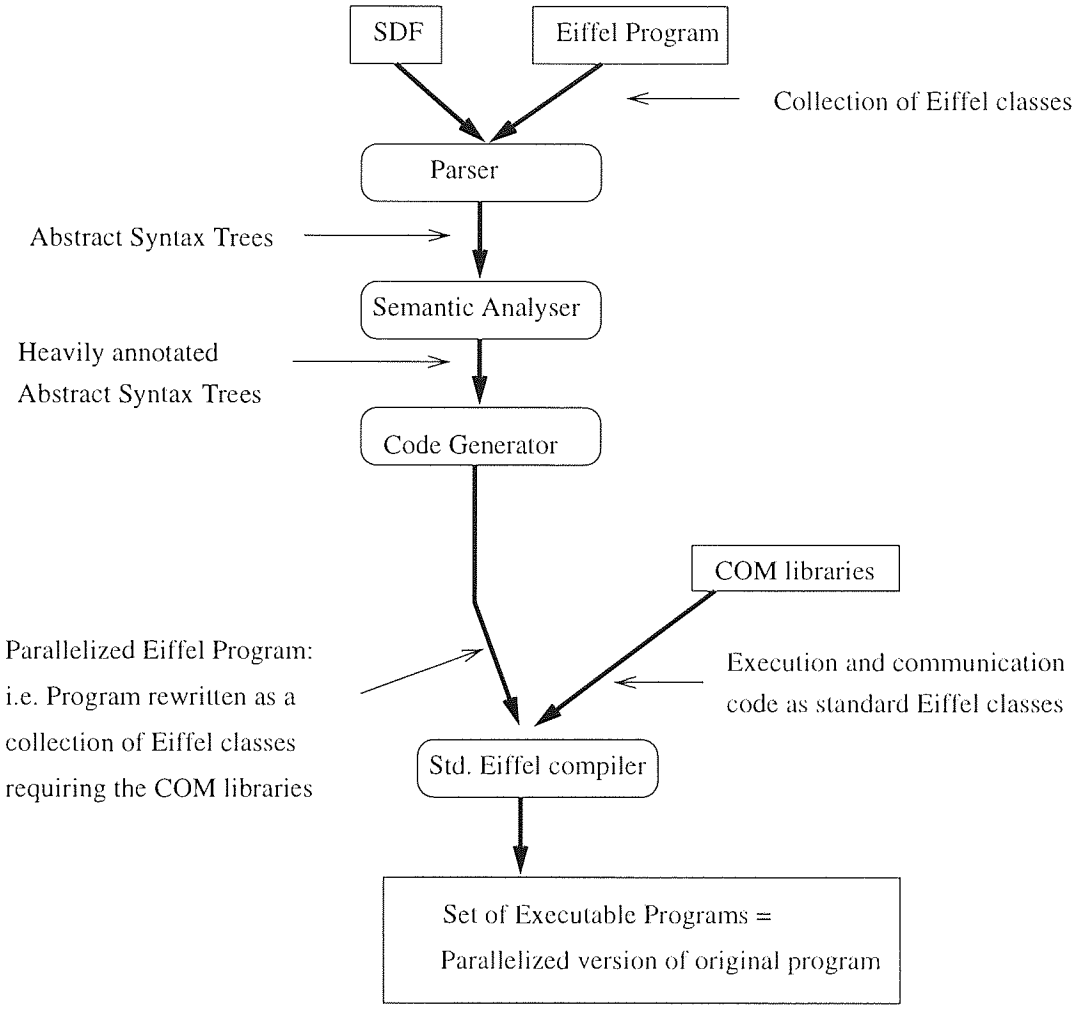


Figure 6.1: Eiffel to Parallel programs

interface of objects which are instances of these classes. Features are composed of a number of parts which define the feature, and so on.

If the grammar describing an Eiffel class is considered, a class - the right-hand side of a grammar production - is describable by a composite of ever finer constructs, with well-defined semantics. Given this view of an Eiffel class (and correspondingly an Eiffel program), the parser as it parses a class will produce an hierarchical structure of constructs which are an abstraction of the input program, assuming there were no syntax errors. This hierarchy is conceptually viewable in two ways:

1. As an abstract syntax tree (AST) where the constructs are objects and form the nodes and leaves.

2. As a construct which is a composite object, where the parts are the expected constructs within the construct being parsed.

These two views will be examined in section 6.2 and 6.3 respectively.

Once the parser has built the above structure the semantic analysis phase takes place. This phase, discussed further in section 6.5, basically takes the transitive closure of most of the program's relationships, evaluating any types that are still unevaluated, and assessing any state change activity: i.e. whether a variable's state is changed in an expression or through being a parameter to a method's application to another object.

6.1.1 Standard Description File

An SDF (standard description file) provides the starting point for the compiler. It indicates the name of the root-class and consequently, in version 2 Eiffel, the name of the file which should contain the root class's definition - i.e. if the root class is named *TEXT* the file is named *text.e*. This file provides directives to an Eiffel compiler including, amongst other things, the following:

ROOT: the root class name;

SOURCE: list of source directories;

EXTERNAL: list of external files;

There are other elements in an SDF, including directives about whether to include assertion checking, debugging, optimisation and/or garbage collection in the compiled program. For the purpose of this work these options are ignored.

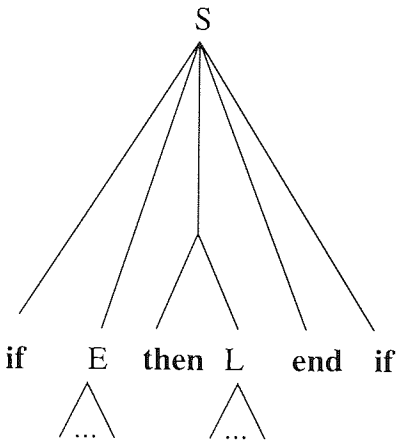


Figure 6.2: Parse Tree for a Simple if Statement

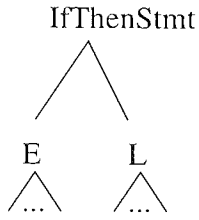


Figure 6.3: Abstract Syntax Tree for a Simple if Statement

The SDF, along with the file containing the root class, provides no explicit information to the compiler about dependencies between classes; these must all be derived by the compiler from analysis of the program text.

6.2 Abstract Syntax Tree

One view of the structure produced by the parser is that of an abstract syntax tree (AST)-like structure. ASTs are parse trees with the syntactic elements eliminated (Fischer & LeBlanc 1988); they can be used as an intermediate form, and annotated, to aid the analytic process. Example diagrams of a parse tree and an AST taken from Fischer & LeBlanc (1988, p 533) depicting a simple if statement (S → **if** E **then** L **end if**) are presented as figures 6.2 and 6.3. The parser produces these ASTs as a result of following a recursive descent style process (Davie & Morrison 1981). The parser follows the grammar description coded into the classes which make up the parser, looking for the next expected construct and placing upon the tree the recognised construct as an object - which is that construct minus syntactic information.

Once produced, the AST can be progressively tagged with attributes that incorporate information about the program being compiled, as might be expected when using attribute grammars[3] (for a discussion of attribute grammars see Fischer & LeBlanc's (1988) book on crafting a compiler where they are attributed to Knuth (1968)). The process of tagging the syntax tree uses the common depth-first left-to-right traversal tree-walk algorithm associated with attribute grammars and syntax trees (Fischer & LeBlanc 1988, pp 514-519). In the terminology of attribute grammars some elements of the objects, representing a construct,

[3]NOTE: No attribute grammar has been written for this work, but the combination of the Eiffel parse libraries and the implementation strategy chosen has the effect of looking like an implementation of an attribute grammar.

are tagged as a result of the synthesis[4] of information from member elements of the construct; whilst other elements become tagged as a result of inherited[5] attributes from constructs of which this construct is a member.

Within this view the AST's nodes are objects which are annotated with inherited and synthesised pieces of information, which differs from the text book description using the language entity alone, as in figure 6.3.

6.3 A Composite Object

Given an object-based approach to structuring the translator there are a number of essential objects which give the translator its shape - these include objects which represent the whole program, classes within a program, features within classes, variables, constants, and the constructs within the program. These objects, created and organised during the parsing phase as a result of the input text, hold information about the specific construct or symbol from which they map. Therefore there are five main types of object which give shape to the translator:

program: holds information about all of the classes used within a program.

construct: there are a variety of these, including if-statements, expressions, sequences etc. They hold semantic information about the programming language construct of which they are an instance.

class: holds all the information about the current class, including what features are exported, what inter-class relationships are used, etc. This is also a construct and therefore inherits the attributes and features of "construct" above.

feature: holds information about features including whether they are attributes, commands, queries, state changers, etc. This is also a construct and therefore inherits the attributes and features of "construct" above.

symbol: holds information about symbols used within features, i.e. variables, types, constants, etc. This is also a construct and therefore inherits the attributes and features of "construct" above.

[4]Synthesis of attributes is the building up of information about a construct using information derived by constituent constructs.

[5]Inherited attributes are those provided to a construct by the construct of which it is a part.

6.3.1 Program Object

The "program object" after parsing contains all the information about the program including class names and associated class information; its visible external behaviour will be similar to that expected from a hash-table.

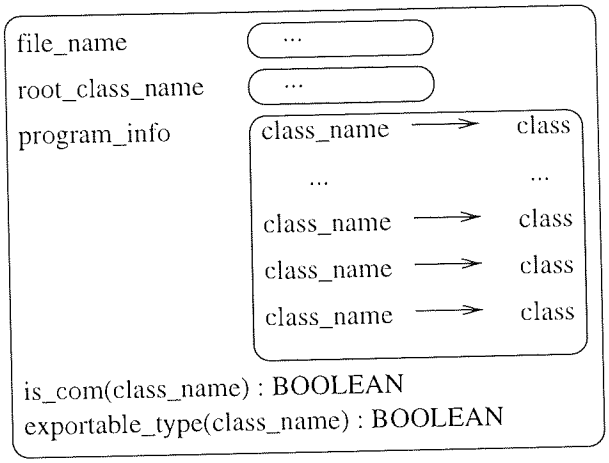


Figure 6.4: Idealised program object

As can be seen in figure 6.4 a program object is made up of at least five elements: the name of the file which holds the main class for this program, the root-class; the name contained within that file, the *root_class_name*; the collection of classes mapped to by their names; and two methods used exclusively in the semantic analysis phase, (see section 6.5) *is_com* and *exportable_type*. It is the creation of this object, and the consequent running of *Create* - not shown on the diagrams as all classes in Eiffel v2 have a *Create* routine - which starts the translation process.

The translation begins, in the *Create* routine, with the program object locating and loading the required root-class, and filling in the appropriate fields. The compilation of the class will be achieved by the creation of a class object; this includes the class-object placing itself into the *program_info*.

6.3.2 Construct Objects

As mentioned previously, the translator is organised around objects. Once below the level of the program-object - i.e. the class objects and their internals - the translator's structure results from the creation of a language construct and its subsequent recognition. All of these constructs inherit attributes and methods of the *construct-object* discussed in this section.

The set of objects which make up a *construct-object* are derived from the Eiffel parse libraries (Meyer & Nerson 1990*a*). A construct can be one of three derived forms - se-

quence, choice and iteration. Using these constructs the translator can define the structure of the language, mapping a grammar production into Eiffel classes, one class per production in the language's grammar. In any construct-object there are at least thirteen features: *parse, production, semantics, pre_action, in_action, post_action, do_print, child_print, pre_print, in_print, post_print, children_print.* These are defined as follows:

parse: as its name suggests, *parse* attempts to parse the current construct defined in the feature *production.* Upon success the "parsed" attribute is set to *true.*

parsed: true, if this construct was successfully recognised.

production: indicates what elements are expected to successfully recognise a current construct. This is redefined in each construct to be an enumeration of the elements on the right-hand-side of the grammar production that describes this construct.

semantics: only called if the construct was successfully parsed, it is defined to perform three methods - *pre_action*, *in_action*, and *post_action*, which are redefined as required within particular construct definitions to achieve a construct's semantic analysis. These three methods, combined, synthesize all the required semantic information.

do_print: if parsing is successful and the semantics are satisfactory then this method performs three methods *pre_*, *post_* and *in_print* which will be defined as required in the current construct. These three methods form the basis of the code generation of the parallelised Eiffel programs.

child_print, children_print: given the hierarchical structure used in the translator, these two features simplify the code generation phase providing useful methods to help code-generate a particular child construct or range of child constructs respectively.

6.3.3 Class Object

A "class object", depicted in figure 6.5,[6] will take as a parameter to its creation routine the name of the class it is supposed to compile, and the currently held *program_info.* A class on being asked to compile itself will check if it has already been entered into the *program_info* table; if it has it can terminate. This effectively avoids cycles in the compilation, which are possible in Eiffel.

The creation parameter *program_info* will be required by the class object as it compiles itself so that it can look up information about parents or clients as required. When a class needs to know about another class it will inspect *program_info*, which will be a hash-table, and

[6] Construct features are not depicted in the figure but are part of a class object.

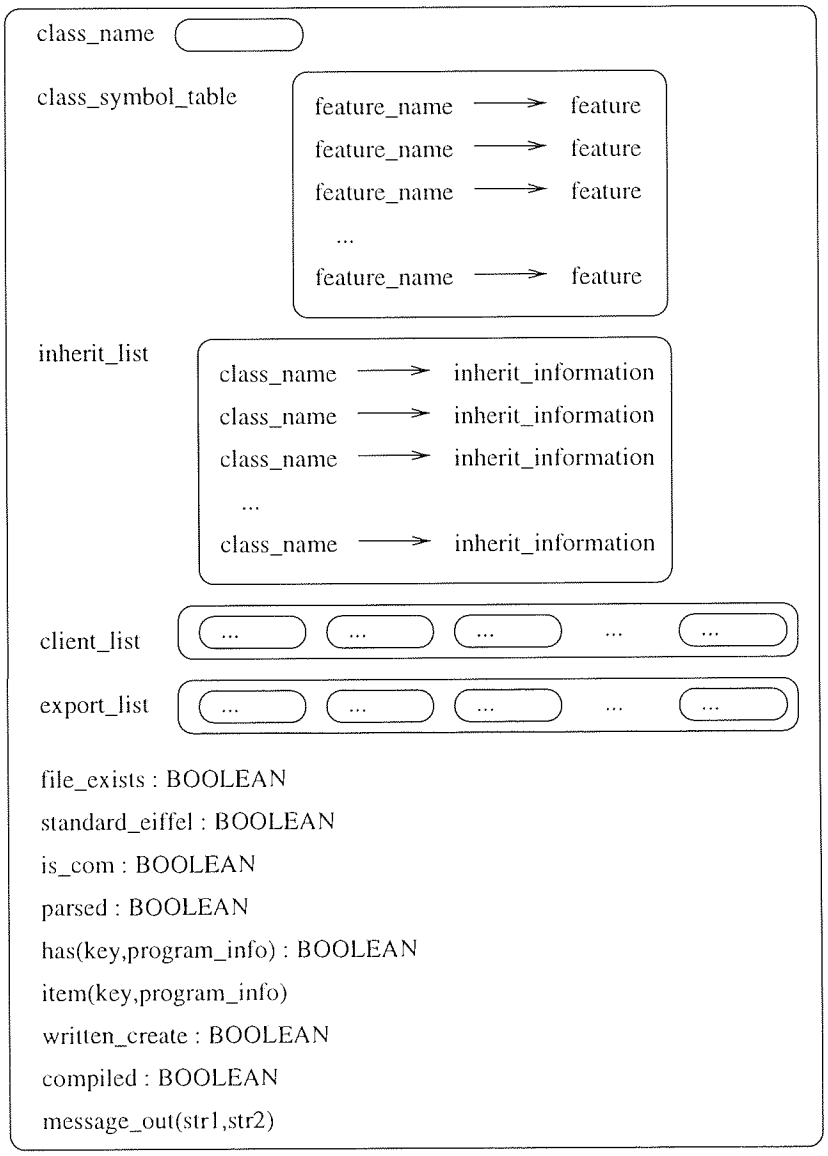


Figure 6.5: Idealised class object

if the class has not yet been compiled, will instigate a compilation, as was seen in the discussion of the program object. It is the passing down of *program_information, class_information, feature_information,* etc. as parameters to creation, semantic and code-generation routines that achieves the tagging of inherited (in the attribute grammar sense) attributes.

A class that needs to be parsed will set up its output files, set up its instance variables, place itself into the *program_info* and then commence analysis.

The analysis of a class will begin with the setting of a state variable to indicate that the class has been or is being parsed; this will be followed by the parsing of the class. After parsing the semantic analysis phase will annotate the class's information including recording the list of inherited classes, repeated export clauses,[7] and exported features. This will include instigating the analysis of any inherited and/or client files which have not already been analysed. The analysis of parent classes will enable completion of a class's export list, as the set of exported features will have now been derived with the expansion of any repeat clauses. The analysis of parent and client files follows the same process as for the current class.

```
analyse_file is
      do
          set_parsed; -- avoid re-parsing classes
          parse;
          if successfully_parsed then
              semantics;
              set_inherit_list(class_inherit_list);
              set_export_list(class_export_list);
              set_repeat_list(class_repeat_list);
              analyse_parent_files;
              analyse_client_files;
              complete_export_list;
              code_generate;
          end; -- if
      end; -- analyse_file
```

Figure 6.6: Analyse a file

The semantic analysis phase consists of finding out all possible relevant semantic information about the features within the current class. This, in turn (as will be discussed in later sections) involves the features in extracting all the relevant semantic information from their elements, providing "synthesised attributes" (Fischer & LeBlanc 1988) to the class.

Code generation is subsequently handled by a class's member features and elements pro-

[7] In Eiffel a short way of saying "export everything that was exported in parent class X" is to say "**repeat** X".

ducing their converted code and by the class object adding an extra routine to deal with message-handling (see section 9.1).

One other aspect that the class object supports, which saves processing time on subsequent runs, is that upon completion of its compilation sufficient information is stored in a "stats" file - i.e. *text.s* - for later reloading, assuming the text of a class has remained unchanged between program compilations.

6.3.4 Feature Object

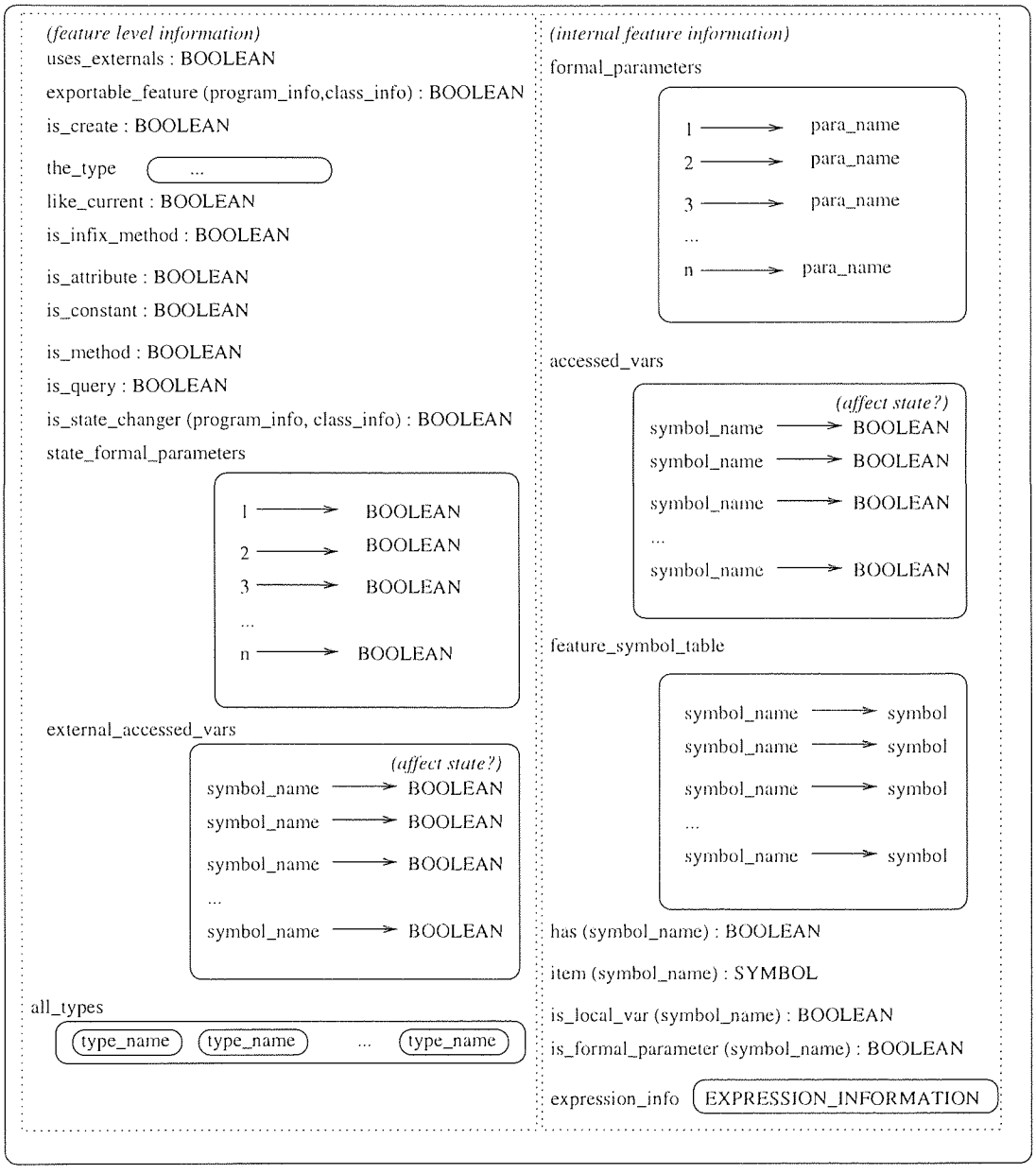


Figure 6.7: Idealised feature object

Features are of one of two categories: either methods or attributes. The compiler treats all features similarly with respect to their names, and upon the parsing of their corresponding definitions constructs a feature-object, populating the object's attributes. The basic structure of a feature object can be seen in figure 6.7. All of the methods and attributes depicted within figure 6.7 are exported by all instances of a feature-object, and are thus made accessible. The feature-object has been depicted as consisting of two parts - feature-level and internal-feature information. This artificial division into two parts highlights the two main uses for a feature-object. The two parts are summarised as follows:

feature level information: information relevant to class level reasoning, synthesised by the analysis process within the feature definition.

internal feature information: information relevant to semantic analysis and code generation when implementing the feature's internal behaviour.

Feature level information

In figure 6.7 the information tagged *feature level information* provides information to the class's objects enabling evaluation of feature properties and interdependencies. For example, *state_formal_parameters* indicates if any formal parameters have their state changed within their corresponding feature definition.[8] The rest of these features are presented in the appendix in section A.1.

Internal Feature Information

In figure 6.7 the information tagged *internal feature information* provides information enabling the feature writing methods to perform semantic analysis and code generation. These features are used by *construct-objects* nested within a feature, for example various constructs will be able to derive necessary information for code generation from that recorded in these fields - i.e. inherited (attribute grammar sense) attributes. These features are presented in the appendix in section A.2

6.3.5 Symbol Object

The "symbol object" is used within feature objects (specifically within a feature-object's *feature_symbol_table*) as a container for assimilated information about a symbol and its associated meaning. As can be seen in figure 6.8 it contains a number of attributes, which are used to derive dependencies and whether or not an object associated with a symbol (in the

[8]This is not ideal, but as with exported features, see section 5.1.2, passing an object as a parameter to a method enables state change of that object if its class exports routines that support write access.

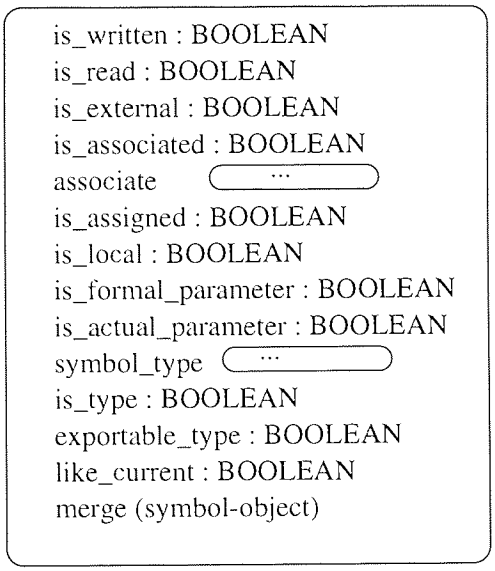


Figure 6.8: Idealised symbol object

rest of this section *symbol* will be overloaded to mean the name and/or the object associated with the name) will need locking during the running of a parallelised expression, etc. For example *is_written* is set if the symbol is the subject of an assignment, or a state-changing method is applied to it. There is a corresponding attribute *is_read* which if true indicates that the symbol has been used in an expression or as a parameter to a method. The other features are defined in the appendix in section A.3.

6.4 Parsing Phase

The parsing phase - i.e. going from the class text to an abstract syntax tree (AST)-like structure - is achieved by the creation of construct-objects, as discussed in section 6.3.2. The "production" feature of a construct object enumerates the expected elements of a particular construct; an example of this can be seen in figure 6.9, which is the implementation of the production for an assignment statement.

The production's return type - LINKED_LIST [CONSTRUCT] - is defined by the parse libraries that have been used within Eiffel. The "once" in the definition means that this method should only be executed once per class (i.e. not once per instance of a class). The execution of this routine builds a parser for this construct which expects an ENTITY followed by the symbol ":=" followed by an EXPRESSION. The "commit" statement within the definition means that having recognised an ENTITY and a ":=" then an EXPRESSION must be next, otherwise a syntax error has occurred. This use of "commit" cuts down on the amount of backtracking necessary: if it has been executed, no other pattern of constructs

```
production : LINKED_LIST [CONSTRUCT] is
   local
      ent : ENTITY;
      expr: EXPRESSION;
   once
      Result.Create;
      ent.Create; put(ent);
      keyword(":=");
      commit;
      expr.Create; put(expr);
   end; -- production
```

Figure 6.9: Example production for an assignment statement

can match the pattern read so far within the current context.

Given an AST-like view, upon execution of *parse* - a member method of any construct-object - and subsequent recognition of the construct, a sub-tree with two children is grafted into the AST: in the example used in this section, an ENTITY and an EXPRESSION (see figure 6.10). The ":=" is syntactic sugaring and thus need not be in the tree.

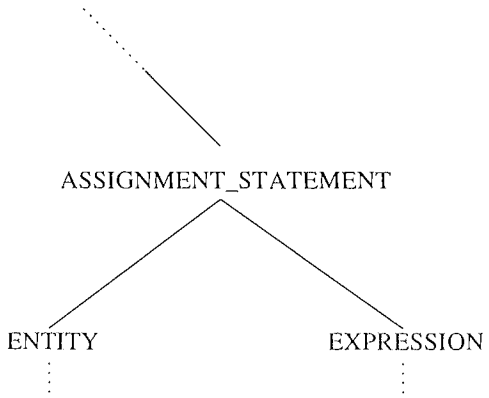


Figure 6.10: Assignment Statement

6.5 Semantic Analysis

The semantic analysis phase begins with the structure resulting from the parsing phase, which is an hierarchically organised entity, viewable as either an abstract syntax tree or as a composite object (see 6.2 and 6.3 respectively). The AST view was a useful perspective for discussion and reasoning about the parsing phase, but for the semantic analysis and subsequent phases the composite-object view aids explanation and comprehension better and thus is used.

The objective of the semantic analysis phase, as mentioned in section 6.1, is to obtain

sufficient information about a program to enable parallelisation. The routines which build up this information are called by "semantics" and are termed *pre_*, *in_* and *post_action* (see section 6.3.2). Put simply, these routines try to extract all of the information they can from the member elements of the construct such that attributes can be tagged in the construct of which they are a member. This includes following through any relationships and aggregating the results, for example to decide whether or not an expression changes the state of an object. The approach of applying a "recursive descent" process to the semantics follows all the way down the construct hierarchy; inherited attributes (attribute grammar sense) are passed down the hierarchy using the technique of "pass by reference" passing *program_info, class_info,* and *feature_info* as appropriate. The synthesised attributes, upon derivation, are made externally available to the construct of which this construct is a member, to be made available for analysis and processing for that construct and the tagging of that construct's attributes.

Methods *is_com* and *is_exportable*

Two routines are made available from *program_objects*, and at various levels below throughout the translator, these are used to ensure the soundness of exportable features with respect to the parallelised implementation. The implementation has inherent limitations on what it can deal with across multiple machines, with respect to inter-machine accessing and information interchange; *is_com* tests that an entity is a concurrent object machine (see chapter 7) and *is_exportable* helps enforce any restrictions upon inter-machine exchange of objects and/or the structure of messages which can be passed between machines.

6.6 Summary

The translator is object-based in structure, viewable from two perspectives as an abstract syntax tree - which helps when discussing parsing issues - and as a composite object built from construct-objects which are in turn built from further construct-objects. The technique used throughout is similar in style to that seen with "recursive descent" compilation techniques with a process of looking in more and more detail at a construct, until recognised or rejected.

The recursiveness of process is continued into the semantic analysis phase where information is passed down to constructs from parent constructs (i.e. inherited attributes), and other information is passed up the hierarchy to parents from child constructs (i.e. synthesised attributes) - as mentioned throughout the chapter this is similar to the ideas of attribute grammars, and would effectively be an operational semantics for such a grammar.

Chapter 7

The Concurrent Object Machine

This chapter presents the design of an abstract machine. The machine provides an abstraction layer above that of a distributed architecture, enabling a straightforward mapping from an object-oriented programming language to the distributed architecture. The machine is designed to provide support for the concurrent execution of programs described in "well written" Eiffel (see section 5.2) and a simple framework for a translator to map onto (see chapter 6).

In this chapter the necessary and sufficient forms of message-passing primitives, which enable utilisation of a general distributed architecture, are discussed. This discussion centres around object-oriented ideas, with these primitives as the only means of inter-object communication.

The ideas presented are based on those put forward in two papers by Gentleman (1981) and Burkowski et al. (1989). The abstract machine described uses the modelling ideas from these papers - specifically Gentleman's (1981) administrator/worker concept - within the paradigm of object-oriented programming.

The abstract machine, developed in this thesis for mapping concurrent object-oriented languages onto, is termed a Concurrent Object Machine (COM).

7.1 An Object

7.1.1 A Reductionist view

Reductionism is the apparent reducing of the complex into simpler independent elements. Taking this reductionist approach to the viewing of an object,[1] a composite-object (i.e. the

[1] see section 2.3.7 for a "standard" definition of "object".

object being viewed) is viewable as an ordered summation (or combination) of a number of independent component-objects which, in combination, working together, give rise to the expected behaviour and properties of the composite-object. For example, a car can be viewed as being a random collection of an engine, wheels, doors, body panels, etc., which when combined produce a concept which is commonly called a car. However, if an object is to be recognisable as a composite-object (e.g. a car) it is necessary that it be "more than the sum of its parts."[2] A simple synergetic view of an object as a collection of parts which in combination, working together, give rise to an expected behaviour is however insufficient; such a view would not incorporate ordering. Without ordering an engine, wheels, doors, body-panels, etc., lying in a heap on the ground would be a car - even though a heap of components lying on the ground cannot usefully function as a car. Therefore to be a recognisably useful composite-object there must be a combination of all the required component-objects, working together, in a prescriptive order - i.e. with some overriding sense of relationship between the objects and some overriding level of control.

7.1.2 Implementing a Reductionist view

To implement the above reductionist view in software the composite-object is conceptually viewed as being a network of communicating component-objects (*Concurrent Object Machine (COM)*). The combination of the communicating component-objects with the addition of an overarching control forms the data and functional space of the composite-object.

The required overall control within an object is achieved by adding an extra component-object to the implementation of the composite-object, termed a *controller*.[3] The *controller* is expected to manage the process of ordering actions, the access to and the locating of component-objects. The *controller* is an integral part of any object implemented as a *COM* and, as suggested by its name, it controls the behaviour of the composite-object's other component-objects. The full behaviour of a composite-object's *controller* is defined during code-generation by the compilation of the source code for the class describing the composite-object, of which the *COM* is an instance. The *controller* "knows" the instance variables and methods for this class and controls method invocation, locking of internal objects for mutual exclusion and external communications. This view of a composite-object as being a combination of a number of component-objects plus a *controller* is depicted in 7.1.

Within a COM, encapsulation is always maintained. Each component-object is told by the *controller* - using message passing - which actions it is to perform. Whilst the *controller*

[2] **Synergism** *n.* the working together of two or more drugs, muscles, etc., to produce an effect greater than the sum of their individual effects. Also called: **synergy**. (The Collins Concise Dictionary)

[3] It is possible that control could be distributed to a composite-object's component-members; this however may prove complex to manage and is not dealt with in this work.

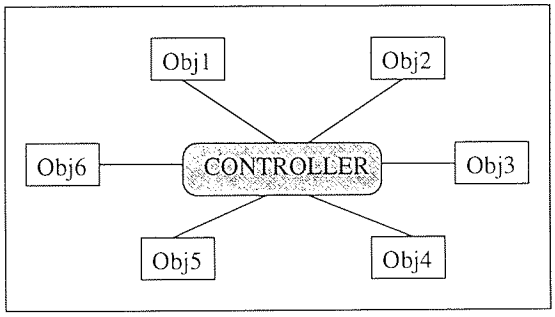


Figure 7.1: A COM

"knows" what messages it can send to the component-objects which together comprise the *COM,* it does not "know" anything about the internals of those component-objects.

Within a COM each component-object, with the exception of the *controller*, is itself viewed as a COM, controlled by a *controller*. However, at the very lowest level of detail objects are not amenable to being described as COMs, because of efficiency or other considerations (e.g. *integers*).[4]

Objects at the lowest level are, instead, viewed as being made up of sequential blocks of code (the methods) with an associated data area (the attributes).

Communication at all levels is achieved using messages; these messages are passed exclusively by message-passing primitives. Great care is required in the derivation of the primitives; they should be both simple to use and theoretically elegant. Ill-thought-out semantics for the primitives, the process structuring forms and the features of the abstract machine can lead to poor functionality, unnecessary restrictions on the achievable parallelism, and possible prohibition of useful programming constructs within a chosen programming language (Brunskill et al. 1995). The primitives and associated model should therefore possess certain qualities; these qualities will affect the ease with which a COM can be used and therefore the ease of implementation of the translator's code-generation algorithms. Qualities required of the primitives include their ease of use; their amenability to efficient implementation, enabling a useful increase in performance by the use of parallelism; and their limited error-proneness in implementation and/or use.

The rest of this chapter looks at the design of these primitives and the abstract machine such that the above qualities are achieved.

[4] As will be discussed in section 9.2.3 a limitation is introduced to make the demonstration of the thesis practical: "every object is implemented as either a COM or is of simple object type e.g. integer, real, boolean, character, or string. This makes the implementation less efficient but does not weaken the demonstration of the thesis."

7.2 The Message-Passing Primitives

The implementation of the message-passing primitives should be such that the design and implementation of the code-generator algorithms in the translator (see chapter 9) are as simple as possible, so that it is easy to reason, debug and refine the models used. The following four concepts must, as a minimum, be supported:

1. Object identification, for objects involved in communication: i.e. the convenience and preciseness of direct naming without the need for mailbox-based implementations (see section 3.3.1).

2. Movement of composite data (i.e. objects) between objects.

3. Synchronization of actions between objects.

4. Multiple threads of control, executing simultaneously, thus achieving concurrency at least.

The message-passing primitives use direct naming synchronous semantics (see section 3.3). The blocking nature of the primitives (Gentleman 1981) provides a straightforward and elegant way to combine message passing with synchronization.

7.2.1 The Blocking Message-Passing Primitives

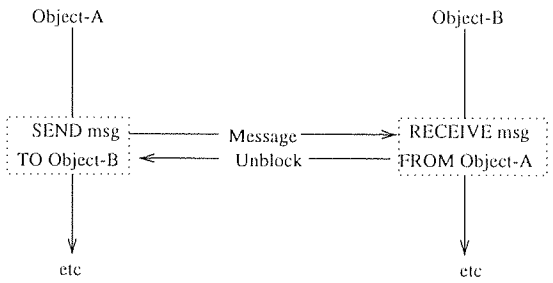


Figure 7.2: Possible blocking sychronization

A *send* primitive is termed "blocking" if it causes the sender to wait, at least until the receiver is ready to receive from it. The *receive* primitive is termed "blocking" if the receiver is required to wait until a message is sent to it. The primitives *send* and *receive* would be termed "non-blocking" if they did not cause a wait until a sent message was accepted, or until a message was sent, respectively.

An example of blocking semantics is depicted in figure 7.2. It shows two objects, *Object-A* and *Object-B*, which execute in parallel until *Object-B* reaches its *receive*, or *Object-A*

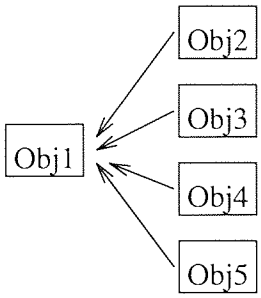


Figure 7.3: Multiple *receives*

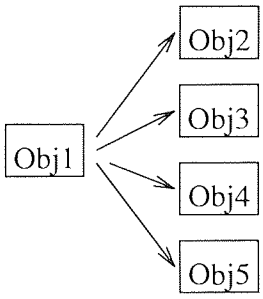


Figure 7.4: Multiple *sends*

reaches its *send;* the object reaching this point first blocks, waiting for the other. When the two objects reach their point of rendezvous the message is transferred and both objects can continue.

The use of a blocking *send* and *receive* can lead to a very limited level of parallelism since an object blocked performing either *send* or *receive* cannot continue usefully. The suggested semantics for *receive* also make it impossible to wait for several messages from different objects; as can be seen in figure 7.2, *receive* is depicted as requiring a definite sender's name upon which it must wait. This is dealt with in section 7.2.2, where a *receive* primitive with "*receive-from-anything*" semantics is presented.

Further, the immediate release of the sender, upon synchronization during the rendezvous, may not provide enough control for the implementor of the code-generation routines in the translator. It may be necessary to cause a sender object to wait until released by the receiver so that an effective implementation can be achieved, e.g. for mutual exclusion, or setting up of some common entity, or indeed to obtain a remote-procedure call semantics (see section 3.3.3). Section 7.2.3 introduces an extra primitive - *reply* - to help deal with this situation.

A problem hitherto not mentioned - the potential for deadlock - will be addressed in chapter 8.

7.2.2 Semantics of the Message-Passing Primitives

In an object-oriented system an object could logically be sending messages to one or more objects, or awaiting messages from one or more objects, which could arrive in an undefined order. The COM's message-passing system should support this logical view of communications; it is depicted in figures 7.3 and 7.4. The operations *send* and *receive* could thus be defined as follows:

send: sends a message *msg* to an object given by *obj* and blocks, waiting until a "reply" (see section 7.2.3) is received:

```
reply := msg.send(obj)
```

receive: blocks, waiting for a message *msg* from the object indicated by *obj*:

```
msg := receive(obj)
```

The above semantics are however too restrictive; it is not possible to model multiple *receives* in an undefined order as suggested by Figure 7.3. Further, it is not possible for an object to receive from an object it does not know about, thus inhibiting dynamic creation and use of objects at run-time.

The simulation of multiple receives

To enable the system shown in Figure 7.3 to be realised, the *receive-specific* semantics require extending to include "*receive-from-anything*":

```
msg := receive(any)
```

The *receive,* with *any* for its identifier, blocks waiting for a message from any object that might send a message to it. The *receive-any* semantics give the benefit of permitting non-determinism in the ordering in which messages can be accepted, thus avoiding an ordering which retards the flow of control inside an object which would be restrictive to the parallelism. The object doing a *receive-any* can be treated like a library subroutine, i.e., anyone can use its services. The enabling of a *receive-any* type of semantics conforms with information hiding principles; even though an object can receive from any other object, objects can only send to it if they "know" of its existence, i.e. it is in scope within the program's source code.

The simulation of multiple sends

The introduction of support for a *receive-any* primitive raises the question of whether a *send-to-all* primitive is required. In practice, a *send-to-all* primitive would add nothing to the semantics of the model as the same result can be arrived at using multiple *sends*. It should be noted, however, that implementation of a *send-to-all* would have a positive effect on both execution speed and orthogonality of the semantics. A form of *send-to-all* is included in the model's implementation with a limitation on what *send-to-all* means. It is restricted to mean all objects that the sender "knows" about, which also helps to maintain conformity with information-hiding principles.

7.2.3 Reply - a useful non-blocking primitive

The form of the *reply* primitive discussed here is as suggested by Gentleman (1981), Cheriton, Malcom, Melen & Sager (1979), and Burkowski et al. (1989). It enables a greater level of control of the concurrency between two inter-related object processes; i.e. it is possible to receive a message and to do some necessary processing before releasing the sender to execute in parallel, e.g. easy implementation of remote-procedure calls (Brinch Hansen 1978).

The Reply Primitive defined

The incorporation of *reply,* a non-blocking primitive that sends messages to other objects, improves the abstraction. Its introduction does not bring the disadvantages of non-blocking primitives, if its usage is restricted.

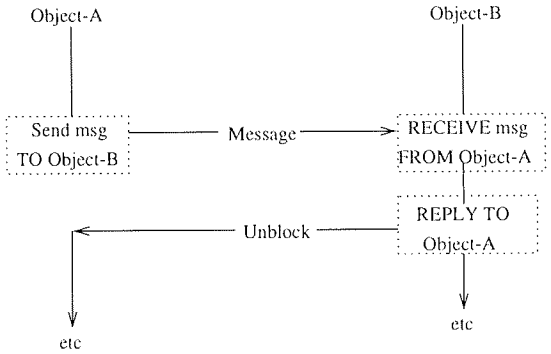


Figure 7.5: The *reply* primitive

The *reply* primitive works as depicted in figure 7.5. The figure shows two objects, *Object-A* and *Object-B*, which execute in parallel until *Object-B* reaches its *receive,* or *Object-A* reaches its *send.* The object reaching this point first blocks, waiting for the other. When both objects reach their respective points the message is transferred and *Object-B* continues. *Object-A* now remains blocked, until *Object-B* sends it a non-blocking *reply.* On receiving this reply *Object-A* and *Object-B* can continue executing in parallel.

The *reply* primitive gives a COM user - i.e. the writer of the code-generator phase in the translator of chapter 6 - more explicit control of the objects they have created. The sender will remain blocked until a sensible time can be found for releasing it and then it is explicitly released by sending a *reply.*

```
msg.reply(obj)
```

The *reply* primitive sends a non-blocking message to the object indicated by *obj*. On receipt, the object receiving a reply should be blocked, having performed a *send* operation. The *reply* primitive does not return a value to the thread executing the primitive.

7.2.4 Identity and the Servicing of Multiple Methods

There are two main issues raised by the incorporation of the *reply* primitive:

Restrictive reply ordering: poor design of the reply and message-passing system may cause an interaction which reduces the potential amount of parallelism;

Object identity too visible: the identity of an object is "power"; given the identity of an object, other objects are free to access and manipulate it, even if this is inappropriate. This could lead to errors not being detected in the writing of the code-generation phase.

Restrictive reply ordering

When a message, which forms a request, is sent to an object, the identity of the requester forms an integral part of the sent request. This loss of anonymity is necessary because it is possible that an object will be performing multiple methods at any instant, thus possibly servicing more than one object at once.[5]

Assuming the simplest of approaches to the servicing of multiple concurrent methods, the replies could be sent in the order that the requests were received, i.e., the object identifier is hidden in a queue, below the level of abstraction of the primitives, so that the *reply* primitive can obtain the object identifier to reply to. This implementation strategy forces an unnecessary restriction onto the system: the ordering of the replies must be the same as the order in which the requests were received.

Consider an object receiving two requests: $request_1$ followed by $request_2$. The requests are being serviced simultaneously; $request_2$ finishes first, because $request_1$ is a time-consuming request. The result is that the *reply* for $request_2$ is held up until the servicing of $request_1$ is finished and the appropriate *reply* sent.

The automatic inclusion of the identity of the requester inside a message, makes the identity available to the servicing object; this removes the artificial need to order the replies. If the identity of the requester is "known" to an object, a message which is a reply can be sent when the servicing has finished: there is no need for an object to wait to perform its *reply* until an appropriate place is reached in its queue. So, the *reply* for $request_2$ can be

[5] Although the multiple servicing of the messages is supported by an object, a COM, for this version of the translator a simpler strategy has been followed giving rise to concurrency through the concurrent execution of member objects only (see chapter 8).

sent, followed eventually by the *reply* for $request_1$. The time between the end of the servicing of $request_2$ and $request_1$ is no longer wasted, as a request similar to $request_2$ can be serviced immediately.

Object identity too visible

The automatic inclusion of a requester's identity could produce a problem. The knowledge of an object's identity is the basic rule for another object to be able to send to it. So any well-defined network could become less secure as identities are passed around at will and unintentionally. The problem is that a rogue object could be written which would "know" too much about the structure and could thus damage the system. Therefore, *identities should be made available very much on a "need to know" basis only* - thus maintaining the principles of information hiding.

This security problem and maintenance of the network's topology is achieved by a slight alteration in how the message is treated. The identity of a sender is automatically included in any message by the *send* primitive. When a *reply* is performed, the *reply* primitive uses the extracted identifier for the object to be replied to within the original request message.

It is still possible to change the topology of the network, but it has to be done deliberately with a standard *creation* routine in the source language or by explicitly passing the object as a parameter to another object. This strategy gives flexibility, allowing for the dynamic creation of objects and a dynamic network.

7.2.5 Summary

The *send* primitive is a function; it sends a message, *msg*, to the object given by *obj* and blocks, waiting for a *reply*. There is an automatic private (i.e. only accessible to the message-passing primitives) inclusion of the identity of the sender within the sent message, for use by the *reply* primitive. The message returned as a reply will be the result of the *send* function.

```
reply:=msg.send(obj)
```

The *receive* primitive is a function which, when called, blocks, waiting for a message from the object indicated by *obj*. The identity of the sender will be part of the received message. Semantics to support *receive-any* are obtained by substituting *any* for the object *obj*.

```
msg:=receive(obj)
```

The *reply* primitive is a command; it sends a reply, *msg*, to the object indicated by the identifier which is a hidden part of the originally received message.

```
msg.reply(original-msg)
```

7.3 Message Format

Although the actual message format is syntactic in nature, it does have implications on the efficiency of implementation, code readability (implementation of the COM) and semantic implications in the way the synchronizing and communicating primitives are viewed and hence used (Gentleman 1981) - see section 3.3.3.

There are two major possibilities - fixed or variable format messages.

Fixed format messages

- **A**dvantages:
  - Efficient implementation (quick in the transfer of messages);
  - Easy implementation;
  - Easy maintenance of the atomicity of the communicating primitives.
- **D**isadvantages:
  - Conceptually inelegant in use;
  - Forces implementation problems onto the user of the COM; e.g. how are longer messages sent?

Variable format messages

- **A**dvantages:
  - Conceptually elegant in use;
  - Easy to understand how to send various messages.
- **D**isadvantages:
  - Harder to implement than the fixed format;
  - Efficiency problems may accrue from the varying lengths;
  - Ensuring atomicity of primitives is a complex task.

The message format chosen for use in the COM is that of variable format due to its more general applicability. Also integral to the message is a field which contains the identity of a sender as required for the *reply* primitive. This identifier is only accessible to the primitives and is not usable by other routines and objects, see 7.2.4.

7.4 Idealised View of a COM-Controller's Objects

7.4.1 The Controller

The COM's (Concurrent Object Machine) controller, termed *controller*, is itself a component-object (not a COM) which "knows" the identity, location and abilities of all the objects in its COM. Its main purpose is to control the behaviour of the COM's objects, filtering and distributing messages and tasks to the appropriate component-objects. The *controller* also sets up communication links between member component-objects (objects local to the COM) and external objects (COMs) when necessary.

The *controller* is itself composed of several objects (termed *controller-objects* from here to clarify the difference from all the other objects involved within the system), all under the control of a *controller-object* called the *brain*. In order that the *controller* may keep efficient and responsive control of the component-objects of the COM in a concurrent environment, it is necessary that the *brain* should never be blocked waiting on a *send* or a *receive-specific* primitive. Equally, any chance for delegation of tasks should be taken. The effect of these two constraints ensures the maximum availability of the *controller*.

Figure 7.6 shows an idealised view of a COM with the *controller* depicted as an entity with connections to all the component-objects in the COM. Inside the *controller* three types of controller-objects are depicted; *brain*, *courier* and *method* objects. The *courier* objects form a group of objects used for communication. They are used by the *brain* to communicate with other objects both internal and external to the COM. The *method* objects correspond with the methods found in the original source code for the compiled program and can each have a thread of control at any moment. The limitations on threads are those necessary to "avoid" deadlock (see chapter 8).

7.4.2 The Brain

The *brain* is the *controller-object* that coordinates the behaviour of the *controller*. As mentioned previously, blocking of the *brain* is to be avoided as much as possible - with respect to *sends* to anything and *receives* from specific objects - the purpose being to maximise the availability of the *controller* and thus control of its component-objects. The *brain's* non-

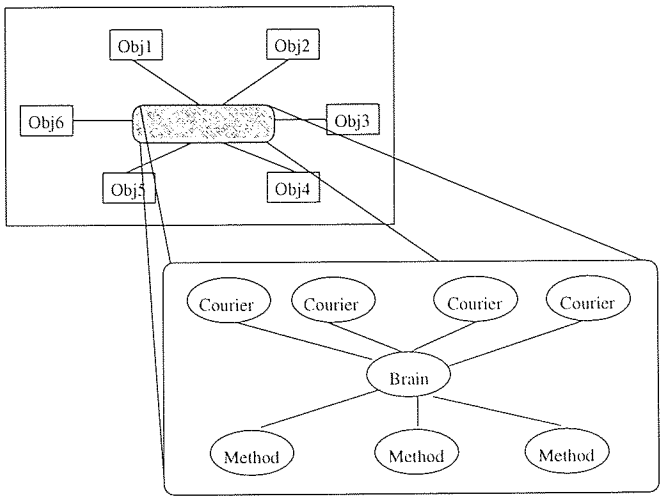


Figure 7.6: A COM

blocking when sending a message is achieved by a variant on the normal communication protocol used within this system:[6] the *brain* uses an asynchronous *send* to achieve all of its *sends* implying no waiting for a reply, and a *receive-any* to do all *receives* implying no blocking on a wait for a response from a particular object. This use of an asynchronous *send* and *receive-any* ensures that the *brain* does not become blocked to the point of ineffectiveness. This protocol does not actually require any new primitives as the *reply* primitive of section 7.2.3 is in practice a non-blocking send, and the *receive-any* has already been discussed in section 7.2.2. This "inversion" of the normal communication protocol is restricted to being used only within the *controller*; all external communications are achieved with the "normal" blocking-based semantics.

The *brain* sets up its communications firstly by creating the COM's component-objects, as required by a class's creation routine. The process of creating a COM-based component-object involves asking a "remote-execute-courier" to perform a "remote execute" of the required object on the required machine, supplying as parameters at least the name and address of the creator. The created object then reconnects back and subsequent communications are, from the point of view of the *brain*, non-blocking, but from the point of view of the *courier* and external viewers of the COM, blocking.

The connection from a created component-object effectively states "I am available for action," the process of connection causes it to block, waiting for the creator to supply it with some action to perform. On receiving a *reply* from the *brain* an object performs its task in parallel to the execution of the *brain* that requested it. On completion of its task, an object,

[6]The normal communication protocol as described in the earlier parts of this chapter is synchronous; the send- and receive-primitives block awaiting each other.

using a blocking *send* to the *brain*, signals completion of the task and its readiness for a new task.

The need to handle dynamic numbers of component-objects within a COM, and set up arbitrary numbers of communication links between component-objects at run-time, impossible to derive at compile time, means that the *controller-communication-objects* in the *controller* must also be created dynamically, as a restricted number will become a "bottle-neck" on the behaviour of the *brain*, and will thus cause blocking. The *controller-communication-objects* are grouped under the name *couriers*; there are basically two similar forms: the first is just a communication channel from a *controller* enabling external blocking behaviour whilst maintaining non-blocking behaviour internally in the *controller*; the second initially creates a new component-object, for a COM, on a remote machine - which could in practice be the same machine - and then behaves as a communication channel.

```
typical_BRAIN
    begin

        initialise_BRAIN  -- set up so that BRAIN can accept
                          -- connections from objects

        initialise_network -- Set up component objects
                           -- e.g. Method objects, couriers
                           -- etc.
        repeat
              request:= receive(any)
              update_state (request) -- records dependencies
                                     -- and method requests
              issue_possible_replies -- initiate methods etc.
        forever
    end -- typical_BRAIN.
```

7.4.3 Couriers

Couriers are a mechanism for achieving the external appearance of blocking message-passing semantics, whilst internally, to a *controller*, permitting asynchronous behaviour. There are two forms: the first causes the remote creation of an object and then behaves as a communication channel; the other just behaves as a communication channel.

All messages are sent to, and received from, component-objects or other COMs by the *brain's* use of *couriers*, which, local to the *controller*, are used to communicate exclusively with objects outside of the *controller* (all *controller* internal communications are done in a pseudo-asynchronous mode using the *reply* primitive). A *courier* upon creation connects to the remote component-object, creating it if it is required to, and then connects to the brain

that created it by performing a *send* stating its availability; it then waits for a *reply* from the *brain* or a message from the remote object for forwarding. Upon receipt of a message it forwards it synchronously to the appropriate object. *Couriers* are created and destroyed dynamically as required.

```
typical_COURIER
     -- simplified view of courier algorithm
     begin
        -- initialise
        create_remote_object -- if required
        connect_to_remote_object;
        connect_to_brain;    -- supply information about remote
        repeat
             message := receive_any;
             if message.from_remote then
                reply := message.reply(brain)
                reply.reply(remote)
             else
                reply := message.reply(remote)
                reply.reply(brain)
             end; -- if
        forever
  end -- typical_COURIER
```

7.4.4 Method-objects

Method-objects are themselves controller-objects and are the implementations of each of a class's methods - i.e. one method-object per method listed in a class. The main reason that a class's methods are broken off as separate *controller-objects* is the need to maintain the non-blocking nature of the *controller*. Incorporation of the original method code within the *brain* would mean that the *brain* would be preoccupied with things other than controlling overall behaviour.

Method-objects fit into the system by being requested to execute themselves given appropriate parameters which are encoded in the message to execute. They incorporate as an attribute the current message received, which is decoded by a routine *pre_method* and then used as the parameters for the actual method (in *do_method*), rewritten to utilise the reorganised object which is implemented as a COM. Finally a routine *post_method* will pass back relevant pieces of information - e.g. results - in the required format. Necessary locking will be requested by the *pre_method*, and unlocking by the *post_method*.

7.4.5 Summary - The Controller Objects

An object (in the general sense of the word) is viewed, for the purpose of implementing it as a COM, as a composite-object, made up of a number of component-objects which are the instance variables of that object's class, plus a *controller*. The *controller* is itself a composite-object (but not a COM); it controls the component-objects under the control of the *brain*. The *controller* contains several *courier* objects which ensure an almost constant interface with the outside world (*couriers* are used by the *brain* to send messages external to the

controller and to await messages and connect objects into graphs, semi-permanently), and *method-objects* which are the implementation of a class's methods.

All component-objects within a COM can execute concurrently. The *controller* with its control code should ensure the avoidance of deadlock (see chapter 8).

The *couriers* are necessitated by the need for the *brain* to remain unblocked and available for administration purposes. They are also necessary to achieve any form of pipelining of objects, whilst maintaining encapsulation of individual objects being connected.

The *method-objects* are necessitated by the need for the *brain* to remain unblocked and available for administrative purposes. They are the implementation of the methods in the original class description, rewritten to deal with the structure of the object as implemented in the form of a COM.

Chapter 8

Deadlock

8.1 Deadlock Avoidance

This chapter[1] looks specifically at the issue of deadlock, in relation to hierarchically composed concurrent object-oriented systems as would be seen using the idea of COMs (see chapter 7). It looks at the issues arising from the use of reductionism and how deadlock can be avoided within this area of automatic parallelism management.

The use of Reductionism - the apparent reducing of the sophisticated and complex into simpler independent elements - aids in the avoidance of deadlock in hierarchically composed concurrent object-oriented systems. That is, the systems of interest are organised and distributed across processors, based upon the compositional nature of objects, and executed in parallel, based upon the concurrency that thus arises.

8.2 The Reductionist View

In science a system is often viewed as a "network" of entities, interacting via various forces and energies. A system can be viewed as being composed of interacting entities. At a further level of detail each of the entities in a system are themselves composed of a system of interacting entities.

This reductionist view of the world, where discrete entities can be investigated and understood in isolation and then recombined to understand the whole, is the basis of much work during the past 200 years of science. Whilst it is clearly flawed when one starts to investigate extremely complex dynamic systems such as the weather (Gleick 1987, Tilby 1992), it has been and still remains a useful tool.

[1] As highlighted in the introduction to this part (page 64) this chapter is as a result of joint work between myself and Prof. C.J.Theaker on an as yet unpublished paper.

This same tool has increasingly become the bedrock of software engineering as software systems are broken into "simple" interacting modules. Indeed, the object-oriented approach may be viewed as an implementation of the reductionist view in the construction of software systems.

This leads to an object-oriented view which puts very rigid boundaries around objects. If formal reasoning is to be both straightforward and valid, a level of encapsulation is required which cannot be broken, where the only means of affecting the behaviour/state of an object is to pass it a message requesting information or telling it to do something - Query or Command (Meyer 1988) - and the object may choose how and if to respond to such requests.

8.3 Hierarchical View of a System

Software systems which are built in a purely reductionist fashion may be pictured using the tree-like structure seen in various methods - Yourdon, Jackson, Rumbaugh, etc. The trees are purely representative of the compositional nature of the system.

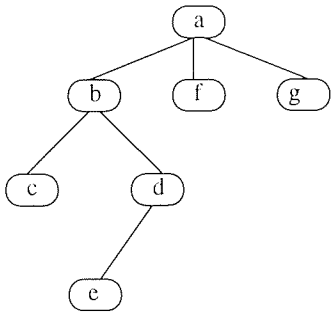


Figure 8.1: A compositional hierarchy

The figure 8.1 indicates that the system (or object) a is composed of objects b, f and g, which communicate by message-passing and are held together by the code of the class which describes the behaviour of class a. However, in knowing a's composition, a will not - and should not - know the composition of b, f or g. Neither should b know about f or f about g, unless that information has been deliberately passed to them.

8.4 A "Pure" Hierarchy or Not?

A hierarchy might be said to be pure if there are no communications across levels, or between sub-trees. This is best seen by two contra-examples:

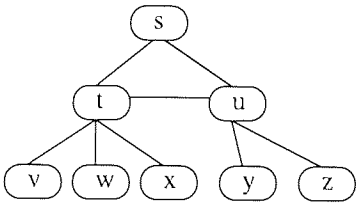


Figure 8.2: Links between subtrees

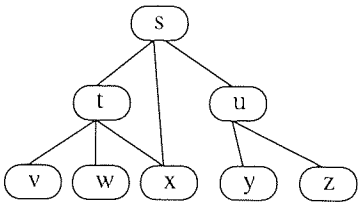


Figure 8.3: Links across levels

1. A link from one sub-tree to another, as t to u in figure 8.2, puts an extra path into the hierarchy and destroys the tree giving instead a cyclic graph. (The edges, on the graph, indicate the ability for direct communication between objects)

2. A link across levels in a tree puts cycles into a system, as s to x in figure 8.3, also giving rise to an cyclic graph.

Both of the contra-examples above give rise to the potential for deadlock and break with the reductionist ideas and techniques, with the consequent detrimental effects on effective reasoning. This of course assumes that any required interactions between t and u of figure 8.2 are achieved by behaviour encoded in s.

Therefore a "pure" hierarchically composed system is one that does not logically have in the composition structure either cycles due to "hooks" of communication between levels, or across sub-trees - i.e. a "proper" Tree.

8.5 Deadlock

Given a tree structure to a system's communications and composition, it is possible to render deadlock impossible. Standard texts on operating systems and distribution by Theaker & Brookes (1993) and Raynal (1988) etc.,[2] list four conditions for deadlock, all of which must be true for deadlock to occur:

[2]Deadlock as a concept was formally introduced by Dijkstra (1968) in his paper on "Cooperating Sequential Processes," the concept's name used in the paper was "Deadly Embrace". However, a one page discussion by Dijkstra (1965) introduced the concept three years earlier with the idea of n processes all trying to access a storage location; suggesting that an invalid solution was one that led to each of the processes saying "After you."

If deadlock is to be dealt with, it is necessary to either avoid it (by design or dynamically) or alternatively to detect and deal with its potential or actual occurrence.

In systems of any size the detection of deadlock at run-time, whether potential or actual, must be dealt with by an algorithm which looks for cycles in a graph and behaves appropriately. Clearly this strategy costs, in terms of resource usage, and could be the cause of problems if the algorithm is not completely correct. When distribution of the elements in a system becomes involved the problem also increases in actual complexity as the shared memory and single processor is lost, with the consequence of an algorithm which is even more expensive in resource terms.

Deadlock prevention can be achieved by ensuring, by design, that at least one of the four conditions for deadlock is not true, i.e. by designing the problem out of the system - the approach used in this thesis. This is a desirable alternative with a parallel hierarchical object-oriented system. The structure of the system with respect to composition is the means by which concurrency is achieved. With suitable source code analysis tools, one can detect if the hierarchy is "pure" and advise of potential deadlock accordingly, and indeed advise on pure design and programming style from the point of view of encapsulation, as this will be the root cause of the hierarchy being broken. This is possible within the Eiffel compiler written for this work. However, it is still necessary to come up with a locking model which is demonstrably (by argument) free from deadlock. The remainder of this chapter devises such a model, resulting in the breaking of deadlock condition 4 above.

8.6 Single-threaded Objects

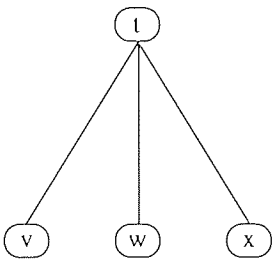


Figure 8.4: Single threading

Each object has a thread of control ("thread" in the operating system sense), such that object t in figure 8.4 can run through its algorithm and by means of RPC with early returns (Gentleman 1981, Burkowski et al. 1989) cause v, w and/or x to perform actions. These

actions in v, w and/or x execute in parallel with the thread in t. The same divisionary behaviour then occurs within v, w and x with successive delegation of tasks to component-objects. The thread in object t would only wait if certain information was required from one of its components without which it could not proceed - i.e. a Query; and similarly for objects v, x, w, if any of them required information from a component. For example:

```
f := w.some_value_request; -- (1)
...
...
...
io.putvalue(f);            -- (2)
```

The statement numbered *2* cannot proceed until *1* has finished because it requires the result of the computation in *1*. So, the set of statements between *1* and *2* can be executed in parallel with *1* - assuming f is not referenced within them. When statement *2* is reached the object must block and wait for the result of *1*.

Given this single threaded delegatory strategy the problem of deadlock does not exist. The second condition for deadlock can never be true in this system. Resources, in this instance objects, are owned and accessed directly by one and only one object with no potential for access by any other threads of processing.

The weakness is that the level of parallelism is not as high as it might be. Objects service multiple and different messages by performing methods. In some cases two or more methods could be executed concurrently, still avoiding deadlock. This leads onto an organisation and distribution of objects based upon composition (see section 5.2.2) but with multi-threaded objects.

8.7 Multi-threaded Objects

Single-threaded objects achieve a level of parallelism but can waste some of the potential parallelism. Therefore there is a need, if parallelism is to be maximised, to allow multiple threads of control in an object, but it is still necessary to ensure that deadlock is designed out. In a multi-threaded object, there would be the concurrency resulting from the compositional structure and RPCs (Remote Procedure Calls) with early returns of the single-threaded objects, but there would also be extra potential parallelism resulting from the simultaneous execution of multiple methods within objects.

Due to the pure hierarchical system structure it is possible to reason conveniently about one arbitrary level of the system, with confidence that the conclusions are valid for the whole system.

Exclusive-rule: No two methods from an object x are allowed access concurrently to a component-object of x.

The *exclusive-rule* is probably stronger than required. It is possible this could be weakened by the judicious use of a Readers/Writers locking strategy that ensures appropriate ordering, fairness and no starvation of methods; allowing at least that multiple methods which only need to do a query could access an object concurrently, assuming no method is writing to it at that moment.

A corollary of the above rule is that no object is free for use until it has finished servicing a message, thus deadlock condition 1 is unbroken (the exclusive assumption). Consequently, objects only become free to service other messages when the methods relating to the current message have finished, so deadlock condition 3 is also unbroken.

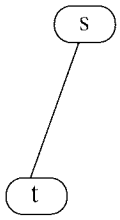


Figure 8.5: A simple compositional structure

Given an object t inside of s, condition 2 states that t is not going to be receiving any further requests from the thread(s) running inside of s until it has finished the original request; given a pure hierarchy, it is only s that can request message servicing. So, the execution of multiple methods with an object cannot occur as a result of external influence.

Given the *exclusive-rule* above, an obvious question is: how is any extra concurrency over that of single-threaded objects achieved? Multiple concurrent methods can only occur as a result of the internal workings of an object, and not due to multiple external requests. Therefore a multiplicity of threads arises as a result of methods internal to an object invoking other internal methods.

In single-threaded objects (see section 8.6), an object's thread runs through, following the normal sequential behaviour found in procedural programs: it starts at the request of a message and any subsequent internal method (procedure) calls are performed by a method call followed by a suspension of the original method until the called method has finished and performed a return - i.e. a subroutine. However, greater parallelism can be achieved if this is modified such that the invocation of a method causes an extra thread to be initiated, which instead of blocking the caller, performs an early-reply (see section 7.2.3) and executes concurrently with the caller.

The result of incorporating the ability to have multiple threads within an object along with the concurrency of single-threaded objects (where parallelism is gained by the delegation of tasks to individual internal objects) increases the level of potential parallelism. However, having stated how extra levels of potential concurrency are to be achieved, it is still not clear how deadlock is to be avoided.

As was stated earlier, neither condition 1 or 3 can be broken therefore either or both of deadlock conditions 2 or 4 must be negated to ensure prevention of deadlock.

8.8 Deadlock Avoidance in Multi-threaded Hierarchical Objects

This section presents one solution to the avoidance of deadlock in a hierarchically composed system of objects supporting multiple threads of execution. The avoidance must be achieved by breaking either of conditions 2 or 4 (see page 97), reiterated below:

condition 2 certain resources are held by processes that are halted, waiting for other resources to become available;

condition 4 there is a cycle in the "wait-for" graphs: this condition implies the three preceding.

The key used to break one of the conditions 2 or 4, above, is the technique used for managing the needed resources (i.e. compositional objects) and their locking for access.

8.8.1 A Method's Resource Requirements

The total of all the objects that a method will need are those it explicitly accesses in its code plus those accessed through the transitive closure of method calls. This is not as large a data set as it may initially appear: it is not necessary to try to lock all of the variables in a component-object that a method is to be applied to because as stated earlier as a corollary to the *exclusive-rule*, an object can only service one external call at a time, thus the component-object would be locked for exclusive access by that method call only (or any methods to which it passed the access lock). The required data can be derived at compile time through the semantic analysis of an object-oriented program, given a suitable programming language (e.g. Eiffel). Notice, however, that this data is not necessarily static. Object-oriented programming involves the dynamic creation of objects during the life of a program, information which may not always be totally dealt with by semantic analysis at compile time,

therefore some processing is required at compile time to ensure that the resource requirements are accurate and up to date.

For each method the resource-access data logically forms a table of objects accessed during execution. It is the use of this information that ensures deadlock avoidance in a multi-threaded object system.

8.8.2 Breaking Deadlock Condition 2 in Multi-threaded Objects

Deadlock condition 2 can be broken by ensuring that a method cannot lock any objects until all of its required objects are free and available for locking. This ensures that deadlock condition 2 is broken: a method cannot hold any objects until it can hold all of them. Thus before any method is executed a check is made as to whether it can run along with others currently executing, i.e. none of the other methods are using objects that it needs. If a method can have exclusive access to all its required objects it will lock them and execute through to completion. This approach ensures the breaking of condition 2 and means that if a method cannot gain at least one of its needed objects, it holds exclusive access to none of them and waits until all become free. The implementation of this strategy for acquiring resources could theoretically be a lookup in tables, checking if each item is unlocked, followed by a locking of all required objects if all are available. It would be necessary that the lookup and locking was atomic as interleaved lookups could lead once more to deadlock condition 2.

However this strategy, whilst effectively presenting a solution to the avoidance of deadlock, introduces other problems. It is necessary to ensure that a method does not starve because of the way in which resources are acquired. If a method needs to lock a number of items this strategy could cycle around giving the lock to various other methods that only need a subset of the objects - effectively locking out (or starving) a process.

A further issue is that the exclusivity of access must be passed to methods that this method calls and this method must correspondingly suspend (This is achievable and is discussed in the next subsection).

In practice, because of the need to have atomicity of the checks for all the objects required and the subsequent locking, plus the awkwardness of the required code-generation in building these tables and the need to deal with dynamically created objects, this approach is not taken.

8.8.3 Breaking Deadlock Condition 4 in Multi-threaded Objects

This condition can be broken by ensuring no cycles are present in the "wait-for" graphs.

The data of resource usage is used here to ensure exclusive access to objects in a pre-specified order based upon ordered unique object identifiers. Thus any method always gains all resources, before starting, in a system-defined order. Therefore a method cannot hold

an object that is required by another method that is executing, as it could not have started without exclusive access. A thread that has exclusive access to all its resources can execute through to completion as it has all the objects that it needs. This is a strategy first proposed by Havender (1968) in his pioneering work on deadlock avoidance. Dietel (1990) whilst discussing this approach states: *"... all resources are uniquely numbered, and because processes must request resources in linear ascending order, a circular wait cannot develop."* Thus this method guarantees the breaking of condition 4 above.

The unique ordering of objects is achieved by making use of the way that the COMs of chapter 7 are implemented.[3] In a running COM-based system each object that is a COM is located at a unique position in the computing space: i.e. each object is on a particular machine (the host) and is connected to via a specific network port (an acceptance port in socket terminology). The pairing of these two pieces of information gives rise to a unique reference for each COM-based object. The ordering is a function which combines the host address with an object's acceptance port (i.e. the place to which another object initially connects in order to achieve a communication link). These references are inherently unique and do not require auto-generating by a numbering program.

The above does not however break deadlock condition 2; for example, it is possible that a method requires objects identified by $object_3$, $object_8$, and $object_{10}$. Currently $object_{10}$ is in use by a different method but $object_3$ and $object_8$ are free, therefore it gains exclusive access to $object_3$ and $object_8$ and waits for $object_{10}$ to become free. This causes no problems for executing threads as they would have gained exclusive access. However, a further method requires $object_4$, $object_8$ and $object_9$; objects $object_4$ and $object_9$ are free but the previous method has already grabbed $object_8$; therefore it gets exclusive access to $object_4$ but cannot gain $object_8$ and must halt upon another process that has its needed resource which is itself halted (deadlock condition 2). Incidentally, this later thread does not get exclusive access to $object_9$ until it has first gained access to $object_8$.

This strategy, whilst ineffective with respect to deadlock conditions 1, 2 and 3, is effective in negating the possibility of condition 4. Due to the ordering in which objects are grabbed and the inability to start until all items are gained, loops cannot occur in the wait-for graphs.

This method, whilst giving an elegantly simple solution for breaking deadlock condition 4, may introduce problems of holding resources for excessive periods of time, whilst executing through an algorithm. Consider a method: it may require access to three objects, requiring a certain amount of time-consuming processing that involves iterating with two of the three objects. The third object may only be needed at the end of the method's processing but,

[3]The implementation of which is discussed in more detail in later chapters, but to clarify the uniqueness of object reference is briefly introduced here.

given the strategy, must be obtained at the start. Thus, $object_3$ is held for an excessive period of time, precluding access by other objects, which leads to possible problems of response time in executing systems. However, accepting this limitation of a lack of timeliness of resource reservation, the method effectively handles deadlock condition 4.

There still remains a problem of how to deal with those methods that should be happening concurrently. The main thread of a hierarchy of threads starts by getting all of its necessary resource requirements; this includes the transitive closure of method call's requirements. Therefore in theory a method called from this thread, which should itself become a new thread, cannot start as it cannot gain access to its required resources until the main method has finished. This is clearly a point of deadlock, as the main method may ultimately need information from the called method. A solution, however, involves passing the right to exclusive access to the called method, as discussed below.

8.8.4 Exclusivity in an Acyclic System

If two methods need to access a common object, they clearly cannot be run concurrently as the coherency of an object's state may be broken, in the ineffective atomicity that may occur if they are both permitted to access the object simultaneously.

Thus if a method m_1 calls (through a transitive closure over calls) a method m_n which requires access to object x, which m_1 has exclusive access to, then m_1 must pass the exclusivity of access to m_n and wait for the exclusivity of access to object x to be returned before it can access object x again; clearly this is when m_n has finished its thread. This does not mean that m_1 and m_n are forced into a subroutining relationship, as m_1 can continue to execute and need only stop and wait when the needed object is currently locked in a descendant thread.

By implication m_1 had exclusive access to everything m_n needed, as m_n forms part of the transitive closure of m_1's calls; also m_1 knows from the table what objects m_n will need and thus knows which exclusive access tokens must be passed to the method. So, regardless of the nesting and structuring of the non-recursive, recursive, or mutually recursive calls, multiple threads can be launched, exclusive access handled correctly and deadlock avoided.

8.9 Summary

This chapter has shown how deadlock can be designed-out of an object-oriented system built purely on reductionist principles. These ideas correspond well with standard Software Engineering practices. It has been argued that it is possible to have deadlock-free behaviour in both single-threaded or multi-threaded objects, if the source programming language encour-

ages good programming style (i.e. encourages the use of these reductionist principles, and is open to straightforward semantic analysis, e.g. Eiffel).

The technique for avoiding deadlock is a standard approach for operating systems, suggested initially by Havender (1968). It relies on breaking one of the four necessary conditions for deadlock; objects (resources in Havender's (1968) work) are locked in a linear fashion based on a system-wide ordering of objects. This locking strategy breaks the possibility of circular chains of objects holding other objects which are in turn needed by other objects in the chain. The system-wide object ordering is inherent to the model and its implementation; it is the quantification of an object's point in "computing space." This number is inherent to all COM-based objects and is inherently unique as only one object can occupy a single point in this "computing space" at any time. The numeric value of the point is defined to be a function from the host-machine upon which the object resides combined with the port at which initial connection is made to enable communication.

The main weakness with the above multi-threaded solution is the amount of message passing that is required. Therefore, future work after this thesis may include determining how some of the theoretical elegancies of reductionism can be broken during the execution phase without compromising the design of the object-oriented system, or indeed its execution - particularly with respect to deadlock.

Another aspect that could be improved in future work is the potential level of parallelism. The strategy outlined in this chapter permits parallelism as a result of internal parallelism within an object, but does not allow for dealing with multiple external requests simultaneously. This would be a beneficial extension as will be seen clearly in the simulation of the standard producer-consumer problem in section 11.4.

Chapter 9

Code Generation

Code generation begins with the data produced by the parsing and semantic analysis phases - i.e. the composite object of section 6.3 tagged with sufficient semantic information to enable parallelisation through code generation. The output of this phase is a rewritten version of the original program able to utilise a distributed architecture.

The chosen code-generation process is "recursive descent" in style with some necessary variations at three specific levels within the translator: class, feature, and expression generation (see section 9.1). The "recursive descent" process applies a method *do_print* to a construct which in turn applies it to its constituent constructs (i.e. the right-hand side of the grammar production describing the construct).

The routine *do_print* is itself made up of three parts: *pre_*, *in_* and *post_print*. The routine *pre_print*, given the parameters of *program_info*, *class_info* and *feature_info*, outputs any information or extra code required before the output of the parallelised version of the construct; this may include unpacking a method's received message. In a corresponding fashion *post_print*, given the same parameters as *pre_print*, outputs necessary code to finish off the code for a construct; for example it will pack the results of a computation and place them into a message for sending back to the caller. The routine *in_print* outputs the construct, including any necessary internal behaviour modifications required for parallelism.

9.1 Variations from "recursive descent" generation

Three of the constructs within Eiffel cannot simply follow the "recursive descent" model of code-generation (at least not when parallelising). These constructs include class, feature and expression. They of necessity incorporate extra code and at least at their respective levels, must redirect the code-generation behaviour. The reason for the deviation from a "recursive

descent" style approach is that for each of these three constructs extra information must be added; i.e. it is not a case of simply translating the member constructs[1] with minor additions: with each of these there are non-straightforward additions and/or rewrites involved.

9.1.1 Class code generation

The translation of a class deviates from "recursive descent" in that a number of extra routines must be incorporated in the code-generated class to enable communication between objects running in parallel. As discussed in chapters 2 and 7, the communication mechanism of objects is that of message-passing; that mechanism, of necessity, was redesigned to allow for parallelisation and distribution of objects across multiple machines. The result of changing the message-passing mechanism between objects is that methods are required within a run-time object to accept and handle these messages. Also within a COM there are other types of message from those specified in the source program; these must be dealt with by the run-time support system. They include exceptions, signals and system status messages. The resulting methods required to aid class code generation include: *produce_class_message_handler*, *handle_message*, and *uncode_paras*. These three methods together write into the generated class a routine called *process_command* which handles all incoming messages (including internal messages e.g. *current.do_x(a,b)*) on behalf of a COM-based object. This extra routine must be generated before the class-code-generation code closes off the class with class invariants and "end-clauses": i.e. not in the *post_print* code-generation phase for the class construct as would be done if it was pure "recursive descent".

The structure of the *process_command* routine is shown in figure 9.1. Placed within a COM-based object, this routine is called every time a message is received; its parameters are the message that was received and the connection from which the request came. The routine checks to see if this is a message (i.e. what might be expected in a normal object-oriented system); if it is, it will initiate the execution of the associated method. If the received message is not a request for execution of a method, but is a reply, or a signal, or an exception then the *process_command* method will pass the message to the standard message-handler for a COM to deal with it (called *c_process_command* in figure 9.1). This standard message-handler for a COM is obtained by making all classes that are supposed to be COM-based descriptions inherit from a class called *COM* (see section 10.2.1).

[1] Member constructs are those terminals and non-terminals that together form the right-hand side of a grammar production.

```
process_command(cmd : like last_command; conn:COM_CONNECTION) is
    local
         local_msg : MESSAGE;
         cstr      : CODED_STRING;
         -- any temporary variables required by the
         -- message handling
    do
         cstr.Create;
         if cmd.is_a_message then
               local_msg ?= cmd.the_message;
               if local_msg.method_name.equal("METHOD1") then
                  -- method1 behaviour...
               elsif local_msg.method_name.equal("METHOD2") then
                  -- method2 behaviour
               ...
               else
                  -- behaviour for undefined message
               end; -- if
         else
              c_process_command(cmd,conn); -- i.e. COM's predefined
                                           -- process_command
         end; -- if
    end; -- process_command
```

Figure 9.1: A Process Command

9.1.2 Feature code generation

The code-generation for a method requires a greater deviation from the "recursive descent" approach than is required for class. As discussed in section 7.4.4, methods including the constructor routine *Create* become objects at run-time. This means that a class wrapper must be generated and a code-generated version of the original feature placed inside, along with any extra routines to enable remote calling and execution of the method. Within the code-generated version of the source class a suitable message handler must be incorporated, and appropriate definitions placed such that the code-generated class of which this method was a member can call this new method and receive any results from it.

The structure of the code-generated class for a method is shown in figure 9.2. The class-name is that of the feature being code-generated. As can be seen it is broken down into three main routines that get redefined: *pre_* and *post_method* and *do_method.*

The *pre_method* routine unpacks a received method request[2] (the message format being that required to send it across a network - see section 10.2.2) into some of the method-object's attributes; these attributes are the formal parameters of the feature's source code.

[2]Method request has been used here to highlight that *pre_method* deals only with requests for method execution; other messages for locking, unlocking etc., are dealt with by the *process_command* routine described in section 9.1.1.

```
class A_METHOD export
      repeat COMMAND
inherit
      COMMAND
           rename Create as c_create
           redefine do_method, pre_method, post_method
feature
      -- local variables from the source feature

      -- formal parameters from the source feature

      -- attributes of the source class that this feature
      -- requires access to

      Create is
          do
                c_create("A_METHOD");
          end; -- Create

      pre_method is
          do
                -- unpack message sent to this method
                -- including actual parameters into formal
                -- parameters.

                -- Set up local variables, i.e. those local
                -- to the source feature code

                -- request class attributes and lock objects
                -- in an ordered way

                -- perform Early Return
          end; -- pre_method

      do_method is
          do
                -- code-generated version of the source
                -- feature, allowing for parameters and
                -- local variables that are now attributes
                -- of this class.

                -- Reply with value of Result if query
          end; -- do_method

      post_method is
           do
                -- pack up and return changed global scalars

                -- unlock all objects locked by this method
           end; -- post_method
end; -- class A_METHOD
```

Figure 9.2: Class for a Method

As well as the feature's formal parameters being made attributes of a feature-class, so also are the locals from the source code for a feature. Also, the attributes from the original class which contained this feature are made attributes of this class, and are initialised during the execution of the *pre_method*. One further part of the *pre_method* is the locking of all the required entities that this feature expects to access; this must be performed in a specific order to avoid deadlock (see section 8.8.3, this is further explained in section 9.4).

The *do_method* routine is a translation of the source feature; it is rewritten to deal with the different environment it is required to execute in, i.e. a COM instead of a sequential object.

The *post_method* packs up any results into a network-transmittable form and sends them back to the caller of this feature. It also must send back any modified entities of simple-data type that are class attributes in the original source code, and unlock any objects that it currently has locked.

The *Create* method from a source class is code-generated in a similar way to other methods, the exception being that because *Create* is a restricted keyword in Eiffel (v2), it cannot become the name of the code-generated class. Consequently the code-generated class which is the parallelised implementation of the source-class's *Create* routine is given the name *MK_class_name*, with *class_name* replaced by the name of the class of which the *Create* is the constructor.

9.1.3 Expression code generation

Expression code-generation abandons the "recursive descent" style completely. Having tagged all attributes during parsing and semantic analysis, within the appropriate objects, and built comprehensive descriptions of any and all expressions, the expressions are rewritten during code generation. Expressions require the addition of a number of extra expression-assignment pairs which are written to deal with method calls within an expression. All method calls are evaluated and the result stored in a local variable before the code-generated version of the source expression is evaluated; the local variable is then substituted for the method call in the code-generated expression. This pre-evaluation of method calls is necessary because of the way in which COMs handle messages (discussed in section 9.2). For example, given an expression:

*e := a + 2 * b.f(c*d.g)*

if *b* is a COM then the message *f(c*d.g)* must be formed into a message such that it can be passed over a network to *b*. Unless the implemented language includes an interpretive element at run-time (within *b*) for interpreting and evaluating *f(c*d.g)* it is necessary to pre-evaluate it. Thus a local-variable is generated and an expression to evaluate and hold

the result of *c*d.g*, and another local-variable to hold the result of *b.f(prev_temp_var)*; at run-time the result in this final local variable is inserted into the modified expression (see figure 9.3 for an example of how this expression may be translated).

```
method_being_compiled (....) is
      local
          l0astype : ASTYPE;
          l1bstype : BSTYPE;
          l2fstype : FSTYPE;
          l3cstype : CSTYPE;
          l4dstype : DSTYPE;
          l5gstype : GSTYPE;
          ...
      do
          ...
          l0astype := a;
          l1bstype := b;
          l3cstype := c;
          l4dstype := d;
          l5gstype := l4dstype.g;
          l2fstype := l1bstype.f(l3cstype * l5gstype);
          e := l0astype + 2 * l2fstype;
          ...
      end; -- method_being_compiled
```

Figure 9.3: Expression Compilation

The generated local variables, as depicted in figure 9.3, are generated using the derived types of the variables and methods: *"l"* is concatenated with a count of the local variables used (ensuring unique variables) and with the derived type of the variable or method.

9.2 The Handling of Messages

The execution framework - multiple distributed COMs interacting by message passing alone with multiple non-COM objects embedded within the COMs - leads to a need to design a mechanism to ensure the successful passing of messages and values between objects at run-time. One of the first problems that this section deals with is how a message such as *b.c.d.e.f* is treated, assuming *b* is an object: is *c.d.e.f* a single message to the object *b*, or is it the message *c* only followed by an application of *d* to the result and then *e* to that result etc?

When sending messages around the system to achieve execution, particularly where objects are logically embedded inside each other (see section 5.2.2), there arises a problem due to the poor support for object integrity provided by most object-oriented programming languages. It is possible in some object-oriented languages (e.g. Eiffel) to do the following:

```
j := e.c.d.set_local_var(z);
```

This is permitting access to and change of an object internal to another object, without "explicit permission" to do so (see figure 9.4); i.e. the export of c, an object in e, is supposed to be available in a read-only mode (at least in Eiffel). However, the above code goes on to read an internal attribute of c and change its state using a method *set_local_var*. Therefore a programmer has broken the "read-onlyness" of the read-only exported attribute. This is against the ideal of information hiding, as indeed is the access to the inside of c which now cannot be safely changed because of possible encoded dependencies in classes that use the class which describes e, due to assumptions that could have been made by the programmer in their decision to implement the above statement. It is however time consuming (in compilation terms) to stop such poor programming practice (this problem of programmer access was also discussed in section 5.2).

It is possible to detect and stop all side-effects in queries (see section 5.2.3) regardless of whether they are at a concrete or abstract level (Meyer 1988). This does not put a semantic limitation upon object-oriented programming languages but forces programmers into a "better" programming style helping them in the localisation of errors, etc. Read-only exported attributes could also be treated as queries within a compiler checking that there were no state changes performed upon the attribute directly or through some transitive closure over message applications, as seen in the above assignment statement.

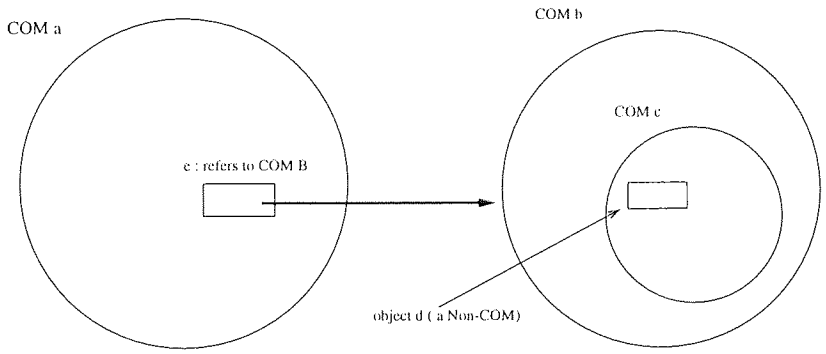


Figure 9.4: Referring to a non-COM from inside a COM

However, given the nature of current object-oriented programming languages, this thesis must consider how to deal with poor integrity of containment in objects; given the theme of the thesis, it is necessary to consider this problem specifically within the context of COMs. There are three possible approaches, each with merit and each with problems. The following three subsections look at the approaches as applied to the above code fragment. The solutions include the splitting/not splitting of a message with extensions to locking and message-

handling within a COM, or the limiting of what can be exported from a COM.

9.2.1 Split the Message

Consider again the statement:

```
j := e.c.d.set_local_var(z);
```

It is possible to split a message into its constituent parts and explicitly manage everything that happens. Assuming that temporary variables are generated for each of the parts of the message (of the form *l_(objects_name)(objects_type)*) and that *request_c_msg* and *request_d_msg* are encoded requests for elements c and d respectively, then the following code fragment could be generated:

```
l_estype := e;
l_cstype := request_c_msg.send(l_estype);
l_dstype := request_d_msg.send(l_cstype);
l_set_local_varstype := l_dstype.set_local_var(z);
j := l_set_local_varstype;
```

The problem with the above code is that *l_dstype* is the object d, possibly a non-COM (i.e. a linked list, record, string, etc.) inside of COM c (refer to figure 9.4) which in turn is inside COM b. Ignoring issues raised by information hiding, COM a, COM b and COM c could all be on different machines and therefore the simple request to *set_local_var* is problematic: the method containing the above code, running inside COM a, cannot see the variable d. Consequently, it must work on a copy, which is not the expected behaviour in Eiffel. Also, whilst working on d, d must be locked from access within COM c, otherwise problems of interleaving may cause its value to be inappropriately read and/or modified.

A solution to the above problem is to ensure that any object implemented as a COM which exports a non-COM must trap all requests to that non-COM and deal with them. This implies that a COM's message-handler must trap and deal with state-changing requests that refer to the internal (possibly non-COM) d, as well as its own expected message requests. It should only be necessary to do this trapping of method calls for state-changing methods, as a "good" query will not change an object's state and thus will give the same result when working upon a copy of the original object. This leads to an awkward compilation process: code must be generated to achieve locking within another object at some arbitrary level of nesting, i.e. embedded in as many layers of object as required; the message-handler code must become incredibly detailed and complex, enumerating all possible message applications to component-objects and their component-objects, and so on.

9.2.2 Do Not Split the Message

Consider again the statement:

```
j := e.c.d.set_local_var(z);
```

It is possible to achieve the expected semantics by delegating the handling of messages to those objects that "know" how to deal with them. The whole of the message is packaged and sent on to the first object. It in turn will extract what it is to do and process accordingly. Given the above expression the code would be of the form given below:

```
-- set m to the packaged up form of "c.d.set_local_var(z)"
l_c := e.com_send(m);
```

The process through which the packed message will go is to be sent to object e; e will unpack it and extract the request for c; the packed message, without the request for c, will then be sent to c - i.e. the packed message now represents *d.set_local_var(z)*. The object c will receive the packed message, unpack it and because d is a non-COM and thus contained within c and accessable, will apply the *set_local_var(z)* to d.

Inside the class describing the COM c, as with the last approach, it is required that all possible requests to object d are dealt with by the message handler. So in generating the code for COM c the compiler must also incorporate code to deal with all of the possible message requests for any embedded objects that are both exported and non-COMs. This implies again a necessity of knowing all messages that an embedded non-COM object can receive at code generation time, as in the last example. The locking is simplified in that an object that contains a non-COM object can more easily manage the locking and unlocking of its attributes than could an external object.

There is a problem that results from this approach: the actual parameters used in the method application, *set_local_var*, i.e. z, may be an arbitrary expression the result of which must be passed through to the object d. Because z is referenced in the class describing the COM a, the variables used will only be in scope within that class, specifically within this feature. The consequence of this is that, as was discussed in the code-generation of expressions (see section 9.1.3), the parameter will be totally evaluated before the message is packed for sending; the resultant value held in a temporary variable will become the item packed for sending as the parameter to *set_local_var*.

The complexity of the locking has been simplified for this approach, but the message handler is required to be as complex as with the previous solution. All possible messages must be enumerated within the message handler so that access to non-COM component

objects is dealt with; this includes enumerating all possible transitive closures over method calls of non-COM objects embedded within non-COM objects etc. For example, imagine the set of messages possible for an object which is a linked list containing arrays which contain strings - the enumeration would be extremely large and wasteful.

9.2.3 A "Draconian" Approach

The third solution considered for dealing with the problems generated by statements such as the one referred to throughout this section is to disallow the export of non-COMs except for simple typed objects such as integers, characters, strings, etc. This may seem a little drastic at first, but in practice it does not put any limitations upon the sort of code that can be written. This decision does have an impact on efficiency, i.e. implementing some smaller objects which have no inherent concurrency as normal Eiffel objects would provide performance benefits.

The restriction is that *"all exported attributes must be either a COM or of a simple type."*

It may seem odd to have a restriction and state that it is not a limitation upon the language. It should however be observed that it would be possible to implement every single object that is not of simple type - i.e. character, integer, real etc. - as a COM: i.e. every object within a parallelised Eiffel system is either a COM or a simple type. The implication of this is that instead of allowing classes to be of mixed types (i.e. some concurrent and some sequential) all classes are compiled into a description for a COM-based object.

This is by far the easiest solution from the point of view of code generation as there is no need to extend any message-handling code to deal with non-COM objects (except simple types), as would have been required in the two previous solutions. Instead, as simple values are exportable and easily encoded in network-transmittable messages they can be dealt with by encoding their handling within the standard message handling and locking of a COM. The message format with respect to the *e.c.d.set_local_var(z)* is that of the previous section (i.e. packaging up the transitive closure of a method application).

9.2.4 Summary

The "draconian" approach is used because of its simplicity and indeed adequacy for demonstrating the ideas discussed in this thesis.

9.3 Eiffel Constructs

This section outlines each of the structures supported within the compiler and the sort of code that should be expected after code generation.

Note in the following descriptions an operation *eval* has been introduced as a shorthand; it implies the breaking up of an expression into all the statements and intermediate local variables that would be required to evaluate an expression - as outlined in section 9.1.3 - i.e. it is not part of the code-generated code but a shorthand for what should be there.

9.3.1 Method application

Consider the method application *obj1.op(e1,...,en)*; a method *op* is applied to a COM-based object *obj*1. The parameters are all expected to be expressions and therefore, as discussed in section 9.1.3, they will each be evaluated and the results stored in local variables which are then packaged up into a message suitable for transmission across a network. The packaged message is then sent to object *obj1*. Example resultant code for this method application is given below:

```
a_method (...) is
     local
          l0e1stype : E1STYPE;
          ...
          lnenstype : ENSTYPE;
          local_msg : MESSAGE;
     do
          ...
          l0e1stype := eval(e1);
          ...
          lnenstype := eval(en);
          local_msg := packaged-up-"op(l0e1stype,...,lnenstype)";
          local_msg.send(obj1);
          ...
     end; -- a_method
```

9.3.2 If statement

```
if cond1 then
   actions1
elsif cond2 then
   actions2
elsif ...
   ...
elsif condn
   actionsn
else
   action-else
end; -- if
```

Using the general if-statement above, the code-generation phase initially will generate code to calculate the result of the expression which forms the *cond1* (the generation is as for

expressions (see section 9.1.3)). The result is held in a local variable which is used as the *condition* in the code-generated code, see below. The compilation of the *elsif* parts follows a similar behaviour: pre-evaluation of the *condition* the result of which is stored in a local variable which is then used as the condition in the if-statement. It should be noted, however, that it is necessary to be careful where the evaluation of the conditions is placed: they cannot all be pre-evaluated before the if-statement starts; some early conditions being true may be supposed to stop the execution of a later condition which would crash the program because of the current state, as is normal practice in programming. Below, the resultant structure is indicated clearly, with the nesting of the *elsifs* one inside the other as if-statements with *else-clauses* which contain the next condition's evaluation and storage in a local variable, and so on. Any else part in the source code is simply the last *else-clause* in the nested if-statements.

```
lcond1stype := eval(cond1);
if lcond1stype then
    actions1
else
   lcond2stype := eval(cond2);
   if lcond2stype then
       actions2
   else
       if  ...
       ...
       else
       lcondnstype := eval(condn);
       if lcondnstype then
           actionsn
       else
           action-else
       end; -- if
       ...
   end; -- if
end; -- if
```

9.3.3 Loop statement

```
from
    initialisation
until
    cond
loop
    actions
end; -- loop
```

Above is an example loop as might be used in Eiffel. Once more it is the evaluation of

expressions that causes deviation from what might be expected. For example the *condition* tested for termination of the loop is pre-evaluated at the end of the initialisation section and stored in a local variable, and also re-revaluated at the end of the loop body. This is shown in the code-generated fragment below:

```
from
     initialisation
     lcondstype := eval(cond);
 until
     lconstype
 loop
     actions
     lcondstype := eval(cond);
 end; -- loop
```

9.4 Locking Generation

Chapter 8 outlined the strategy to be applied to achieving mutual-exclusion by locking and how deadlock has been designed out of this COM-based model of concurrency. This section details how the locking can be implemented within this framework. As discussed in section 7.4.4 and depicted in figure 9.2, locking is requested by the *pre_method* of a method-object which is an instance of a feature-class, and unlocking is requested by the corresponding *post_method*. The locking and unlocking must, as required by section 8.8.3, be in a specific order to ensure the breaking of what was termed deadlock-condition 4 - "there is a cycle in the 'wait-for' graphs" - i.e. there must not be any cycles. These give rise to a number of questions: where is the lock held and what manages it; how is a lock actuated; how is a lock released, and how is a lock passed on?

9.4.1 Lock location and control

The managing of the locks is of necessity placed with the object that is to be locked, i.e. an object deals with its own locking. Upon creation the owner of the lock is the object that created it.

LOCK When a method requests exclusive access to an object, the object will verify whether the request is valid (i.e. the object is not already locked); if the request is valid the lock will be given to the requester; if however the object is already locked the request will be queued and thus the requester will block.

PASS_LOCK Locks are passed on during calls of methods: a method-object mo_1 has a lock on an object; it calls another method-object mo_2 that needs the locked object;

mo_2 is passed the lock by sending a request of mo_1 in its *pre_method* asking for the lock to be passed to itself; thus, gaining the lock, it can proceed. This is achieved by the currently locked object receiving a request to "pass-lock" - i.e. to pass the lock on to another object - the locked object checks if the originator of the request has the right to do this (i.e. it has the locking object's 'key'). If so it will pass the lock, stacking the previous locking object mo_1 so that on release by mo_2 mo_1 is given the lock back.

UNLOCK When a method is finished with an object that it has locked it simply sends an unlock to the object. The object checks first in its stack for an object that passed the lock to this one and gives the lock back to that object, if present; if the stack is empty the lock is given to the next request, held in a queue. If both the stack and the queue are empty then the object is unlocked.

This implementation strategy achieves the requirements of chapter 8 of no deadlock and as will be seen in sections 11.5 and 11.4, does not suffer from starvation.

9.5 Early Return

One aspect of the calling of a method, which has not been discussed, depends upon the previous section 9.4, that is, how early returns fit into the execution model. An early return is needed to enable a called and calling method to continue in parallel, the early return releasing the caller to continue. It can be seen in section 9.1.2 that a method gains exclusive access within the code-generated *pre_method* to all of the entities that it needs exclusive access to. This locking is done in a specific order to avoid deadlock and the method cannot continue to execute until it has gained all of the required locks. Therefore once the locks are gained there is no reason for the caller to remain blocked, it cannot after all read or alter any of the objects it shares with the called method as they are locked by this called method. Consequently an early return can be performed at this stage releasing the caller to execute in parallel.

This approach is reasonable even for queries, as long as the subject of any assignment involving a query is locked. Consider the following code fragment from a caller of a query q:

```
y := q(a,b);
...
...          -- code which does not access y
...
if y = z then
  -- ...
```

In the above code y is the subject of an assignment, the result of executing the query q. However y is not accessed in between the initial query request and the if-statement; consequently the caller could execute in parallel with the query calculation, as long as the value is placed into y when returned and the if-statement does not proceed until y has that value. This is easily achieved by locking y as though query q requires exclusive access to it, placing in a table a record of the assignment which is performed along with the unlocking of y when the result is returned. This stops any accessing of y until the required value has been assigned and thus helps ensure the expected sequential semantics, with parallel execution.

9.6 Inheritance

The simple analysis of code-generated files is handled by the parser, as shown in figure 6.6. A class, "realising" that it inherits another class will initiate the analysis of that class. However, classes must be made to reliably inherit from the *COM* class (of section 10.2.1; also see the end of section 9.1.1) either directly or indirectly. This requirement is because all objects described by classes must be COM-based, i.e. must inherit from the COM classes, as the only supported objects are of simple type or COM-based (see section 9.2.3).

Consequently, the top class in an inheritance hierarchy must be made to derive from the *COM* class. Also the code-generated *Create* routine for any class must call the code-generated *Create* of its parent, ensuring that all class attributes have been appropriately initialised. This ensures that all classes have access to the appropriate COM-based routines.

As a result of making the top class in an inheritance hierarchy inherit from class *COM* there is another problem to solve. If a class uses multiple inheritance, and each of the parent classes in turn inherit from the *COM* class, there arises a problem of name clashes in the class which is doing the multiple inheritance. In practice this is quite easily resolved: within a code-generated class that is using multiple inheritance, within the inheritance clause of all but the first class (all classes inherit from a COM class) code-generate a complete rename set, renaming all name clashes away.

One further issue that needs to be addressed is the potential clash between inheritance and concurrency, initially referred to in the literature as an *inheritance anomaly* (see Matsuoka, Wakita & Yonezawa's (1993) original paper on the subject). This inheritance anomaly was initially described as a conflict between synchronization and inheritance. It has however been suggested through McHale's (1994) PhD thesis that the apparent conflict is not actually a conflict between synchronization and inheritance, as much of the literature in this area suggests, but is because of the conflict between inheriting a mixture of sequential and concurrent code (i.e. code with synchronization properties). Given either explanation for

the apparent anomaly, the implementation within this thesis was thought to circumvent the problem.[3] If Matsuoka et al. (1993) and the derived literature are right then the implementation here avoids such a problem because a programmer cannot modify the synchronization mechanisms and behaviour as it is out of their control and consistent across a program. If McHale (1994) is right then, given the restriction of section 9.2.3, all code-generated classes are in essence concurrent and there is no mixing of approach within the inheritance hierarchy. However McHale goes on to state that the problem is insoluble and that all one can do is minimise the symptoms. Given the complexity of the semantics involved and McHale's (1994) work it cannot be stated with any confidence that there is not an inherent problem. The investigation of this is therefore left to be a post-thesis activity.

9.7 Summary

Code generation, as with other processes within the compiler, follows a "recursive descent"-style process; this arises from the structure of the trees which resulted from the translation of programming language constructs. This simple recursiveness of process is broken by three language constructs: classes, features and expressions, where extra routines are required, or extra classes, or extra statements and local variables are required respectively.

The handling of messages, given the distributed nature of the resultant implementation, requires thought as to how a message such as *"j := e.c.d.set_local_var(z);"* should be compiled. The two main solutions are:

1. break up the message and send the first part to the first object (i.e. send c to object e); wait for the result; then send the next message d to *lcstype*, i.e. the result of sending c to e; wait for the result and so on until the final result is obtained.

2. send the transitive closure over message calls, as a packaged message to the first object, i.e. e.

The second approach implies simpler mechanisms with respect to variable locking inside COMs. However, as was observed within section 9.2.3, it would be reasonable to suggest that all objects must be one of only two formats: either COM-based or of simple type. This does not decrease the power of the language and is sufficient for demonstrating the thesis (as pointed out it does however have efficiency implications). One obvious implication is that a COM object can only export COMs or simple-typed objects. The consequent message-passing and handling mechanisms are thus much simpler.

[3] An area of future work: demonstrate by proof, using a semantics for inheritance, that there is or is not an anomaly with the use of inheritance within the implementation used for this thesis.

Part III

Implementation

Introduction to part III

This part looks at the implementation of the solution presented in part II to deal with translating a sequential object-oriented program in Eiffel into a parallelised program. Chapter 10 presents an overall look at the implementation.

Chapter 10

Overall Implementation

This chapter discusses briefly the work done in implementing the ideas contained within this thesis.

The structure of the compiler arose because of a view of constructs as composite-objects (see section 6.1) which have components, the components being the non-terminals and terminals that would appear on the right-hand side of a grammar production describing a construct. It is this structure that leads to a "recursive descent" style of processing in each of the phases: a method is applied to the top construct (a class) and then (whether parsing, analysing semantics or code-generating) recursively within its component constructs.

10.1 The Translator

The implementation of the translation process made use of the "lex" and "parse" clusters of ISE Eiffel and also the standard data-structure cluster. This minimised the amount of code that needed writing from scratch, enabling concentration on the more awkward aspects of compilation in relation to parallelism.

The two clusters "lex" and "parse" provide a framework for building scanners and parsers for LL[1] grammars. The parse cluster provides support for *backtracking* and a routine called *commit*. The *commit* routine avoids the need for exhaustive searches when syntax errors occur, i.e. if a compiler writer knows at a certain point in a production's description that no other production is of a similar pattern then they can incorporate a call to *commit*; this call implies that if any part of the remaining construct is not found, a syntax error has been detected. An example of *commit's* usage was shown in figure 6.9.

[1] LL : Left context, leftmost non-terminal expanded first, no left recursion.

10.1.1 Scanner Implementation

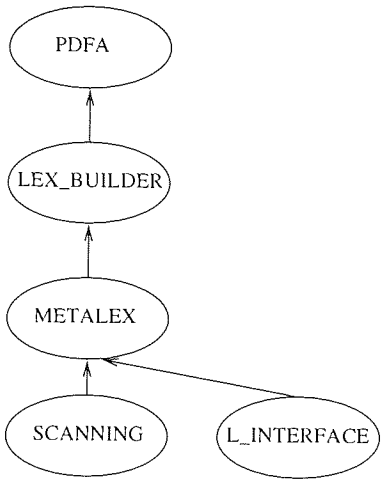


Figure 10.1: Lexical class structure

A simplified structure of the classes for the lexical analysis part is presented in figure 10.1; the lexical analyser implemented for this thesis inherits from the *L_INTERFACE* and enumerates all of Eiffel's keywords supplying a token type for later recognition by the parser, and the expressions that describe all possible token constructs - e.g. for integer constants, characters, strings, keywords, etc. - again with a token type for later recognition and use by the parser. None of the above classes in figure 10.1 needed any modification to implement the lexical analyser.

10.1.2 Parser Implementation

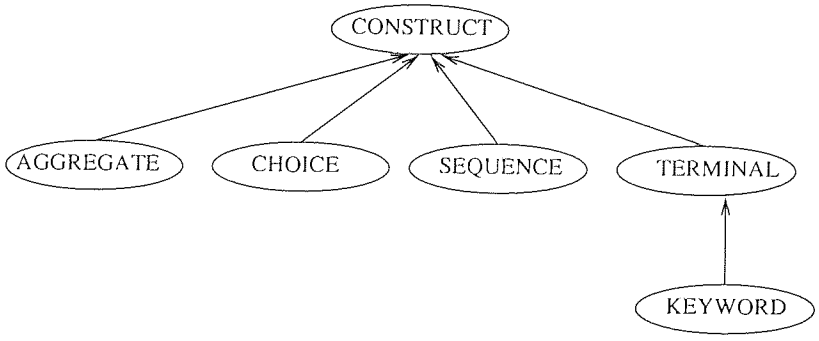


Figure 10.2: Parse class structure

The supplied parse classes provided a solid base upon which to write the parser (see figure 10.2). Given the chosen method of implementation (see chapter 6) the ideas presented in *CHOICE*, *SEQUENCE*, and *AGGREGATE* were, in the course of implementation, each ex-

tended by the use of inheritance - the new classes were called *C_CHOICE*, *S_SEQUENCE*, and *A_AGGREGATE* respectively. The extensions included incorporating basic routines to simplify code generation, i.e. for most of the Eiffel constructs the code generation routines added to these classes could be used directly, as inherited, with no modification. *S_SEQUENCE* was further extended to incorporate the idea of allowing a separator to be used as a terminator, see the description for this class, called *SE_SEQUENCE*.

The four classes and their equivalent grammatical ideas are:

C_CHOICE is used if there is an option on the right-hand side of a production, e.g. A is b or c or d, "`A ::= b | c | d`".

A_AGGREGATE is used if there is a requirement for one construct to follow another, e.g. B is the construct E followed by F followed by G, "`B ::= E F G`".

S_SEQUENCE is used if there is a requirement for iteration, but the seperator indicates that another element of the construct must follow, e.g. "`H ::= h ";" H | h`".

SE_SEQUENCE is used if there is a requirement for repetition and the separator, ";", is allowed to terminate the sequence, e.g. "`J ::= j ";" [J] | j`".

Each of the constructs within the compiler inherit from one of these classes, adding their specific production descriptions (see section 6.4) and refining any methods as necessary.

10.1.3 Recursive Class Relationships and Compilation

As outlined in figure 5.1 it is possible to have recursive relationships between classes; thus there is a need to be able to handle this within the compiler. Unlike C which typically, at least on UNIX, uses a *make* file (an algorithm to tell the system how to compile the programs), Eiffel compilers need to obtain the relationships between classes by derivation.

The request for compilation of a class, whether it is a client or is inherited, goes through the following process:

```
if not program_object.has(class_to_compile) then
        a_class_object.Create(class_to_compile,program_object);
end; -- if
```

The above pseudo-code checks whether or not *program-object*, as in section 6.3.1, currently holds a *class-object* for *class_to_compile*, i.e. if present, it has already been compiled or is in the process of being compiled. This neatly avoids any infinitely recursive compiles resulting from classes which refer to themselves as clients either directly or via mutual-recursion. Also,

this avoids constant recompilation of classes used throughout the program. Note also, it is the responsibility of a *class-object* to "place" itself into the *program-object.*

The compiling of a class within a *class-object*, which results in the code-generated code, follows the process below:

```
parse;              -- This is performed inside the class-object
set_parsed;         -- Indicate for any following attempts at
                    -- compilation that it has been parsed
if parsed then
                    -- if it successfully parsed
        ...
        semantics; -- attribute the abstract syntax trees
        ...
        analyse_inherited_files;
        analyse_client_files;
        ...
        code_generate;
        ...
else
        raise_syntax_error("FILE UNPARSED");
end; -- if
```

The process of analysing each of the inherited and client classes follows through the process of checking if each class has been compiled and if it has not, compiling it.

10.2 Concurrent Object Machine

The implementation of the Concurrent Object Machine (COM) (see chapter 7) made use of the "network" cluster designed for network programming within Eiffel (Hillman 1990). Hillman's (1990) classes are organised around an inheritance hierarchy which provide a client-server abstraction for network programming with objects. The simplest object, a client, described in the class *NET_NODE* is able to connect to "server" objects and exchange messages; inheriting from this class, *NET_ACCEPTOR* objects can both connect to "server" objects and accept connections from other objects; descendent from this class *NET_CMD_PROCESSOR* provides an abstraction for connecting and accepting connections and also support for the provision of a service, as would be required by any "server" object.

As well as providing a set of clusters to model the type of objects required in a client-server environment, the network cluster also includes a class, *NET_CAPABLE*, which provides an abstraction for messages that pass between network objects. A further class discussed in Hillman's (1990) paper, *FORKABLE*, provided support for forking processes; however, this class was removed from the distribution because Hillman (1990) realised that non-blocking input/output on UNIX's stdin/stdout respectively could be achieved without forking separate

processes. Two other classes were provided, *NET_CONNECTION* and *NET_CONN_LIST*, which respectively provide an abstraction for connections between objects and a connection holder for a collection of connections.

The ideas presented within Hillman's (1990) paper, specifically the organisation of the classes within this "network" cluster, together provided a good base for implementing the concept of a COM. However with the exception of two classes (*NET_NODE* and *NET_ACCEPTOR*) all of the classes needed to be completely rewritten. The resultant COM classes are (as divided up in Hillman's (1990) paper) split into three groups: *net-nodes*, clients and servers; *net-capable*, objects that can be passed over a network; and *net-connections*, which models the connection of one *net-node* to another.

10.2.1 Net Nodes

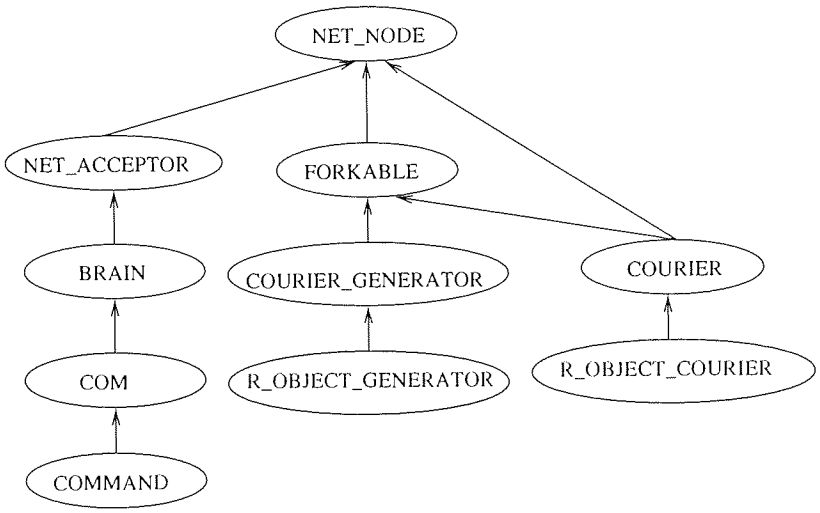


Figure 10.3: Net-node inheritance hierarchy

The classes in the system are as follows:

NET_NODE: Abstraction for implementing client-server type architectures, it provides the idea of a client object able to connect to and communicate with a server object.

NET-ACCEPTOR: Descendent from *NET_NODE*, it is the abstraction for implementing server type objects; instances can make connections to server objects as well as being able to accept connections from either server or client objects.

BRAIN: Descendent from *NET_ACCEPTOR*, it is similar in architecture to Hillman's (1990) *NET_COMMAND_PROCESSOR*, except it also incorporates the ability to fork

processes, which is a requirement with COMs. Descendent from this is the *COM* class which encompasses the whole idea of a COM-based object.

COMMAND: Descendent from *COM*, it is inherited by all method-objects (the implementations of a class's methods within a COM, see section 7.4.4). It provides the required structure for the classes which implement each of the method-objects including *execute*, *Create* and *process_command*. Also the *COMMAND* class specifies, using deferred routines, that a class implementing a method-object should define *do_method*, *post_accept* (specifies what should be done if an object connects to this method to use its service), *pre_* and *post_method*.

FORKABLE: This class provides the ability for an object's thread of execution to split into two processes.[2]

COURIER: Descendent from *NET_NODE* and thus able to exchange messages and connect to objects capable of accepting connections. In practice a *COURIER* is created by either a *COURIER_GENERATOR* or a *R_OBJECT_GENERATOR* (see below). A courier once created will, if it is an *R_OBJECT_COURIER* create the remote object requested by the *BRAIN*. It will expect to receive from the remote object its acceptance port and will then connect to it, providing the identity of the *BRAIN*. After successfully creating and connecting with the remote object the courier connects to the *BRAIN*, providing the identity of the remote object. After this is done the courier goes to sleep until either the *BRAIN* needs to send a message to the remote object, or the remote object needs to send a message to the *BRAIN*.

COURIER_GENERATOR and R_OBJECT_GENERATOR: Descendent from class *FORKABLE*, instances of these two classes, when asked for a new courier, will fork into two processes; one process produces either a new courier or a remote-object-courier respectively, and the other process returns control immediately to the caller. The *BRAIN* within a COM contains one instance each of *COURIER_GENERATOR* and *R_OBJECT_GENERATOR*; each time it requires either a new courier or a new object on a remote machine it requests the item from the appropriate generator.[3]

[2] Threads, which as light-weight processes would be more appropriate, were not supported in the implementation environment for most of the development time of this thesis and hence are not used.

[3] "Remote" includes both proper remote machines and the same machine.

10.2.2 Net Capable

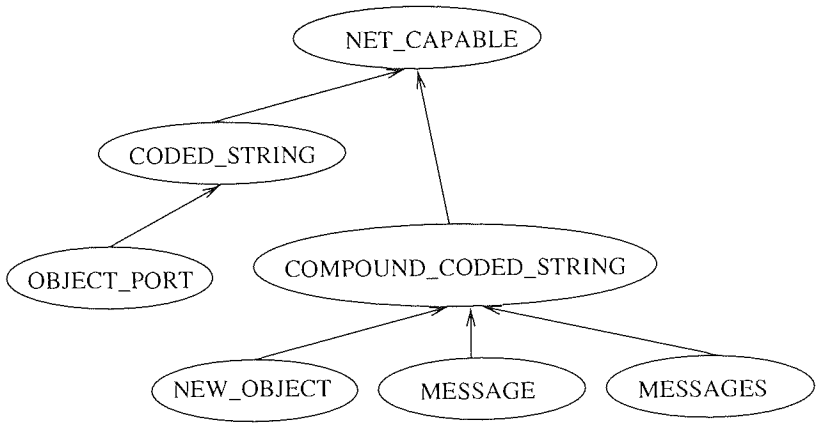


Figure 10.4: Net-capable inheritance hierarchy

NET_CAPABLE: An abstraction for implementing objects that can be passed over a network, *NET_CAPABLE* is used within the *NET_NODE* class hierarchy when information is to be exchanged between COM-based objects. As with the *NET_COMMAND-_PROCESSOR* of Hillman's (1990) *NET_NODE* hierarchy, this class did not incorporate the right level of abstraction, i.e. it did not provide enough flexibility to support the message structuring required in this thesis, hence it was rewritten. All of the classes descendent from this class can be passed across a network.

CODED_STRING: Descendent from *NET_CAPABLE*, it incorporates the data items that can be passed across a network. Instances of it can contain simple integers, characters, booleans, strings, etc., or suitably encoded instances of classes descendent from *NET_CAPABLE*; this implies that all the classes within this hierarchy, with the exception of *NET_CAPABLE*, require an exported routine *coded_string* which provides the *CODED_STRING* form of the class.

OBJECT_PORT: Transmittable across a network as a *CODED_STRING*, it is used to represent a distributed pointer, i.e. a machine and port number providing a unique address for an object, to which other objects can connect.

COMPOUND_CODED_STRING: An aggregation of *CODED_STRING*, enabling compound messages to be exchanged between objects.

NEW_OBJECT: An aggregation of the *OBJECT_PORT* class and an object name, the name being the identity of the object. This name is useful in the debugging of parallelised programs; at run-time status messages (if turned on) will indicate the originator

of a signal, message, or exception, providing traceability in the execution behaviour of a parallel program.

MESSAGE: Abstraction of the types of message that one object sends to another within a standard Eiffel program.

MESSAGES: An aggregation of a number of *MESSAGEs*. It provides support for the passing around of a compound of *MESSAGEs* like: "`c.d.set_local_var(z);`" from section 9.2.

10.2.3 Net Connections

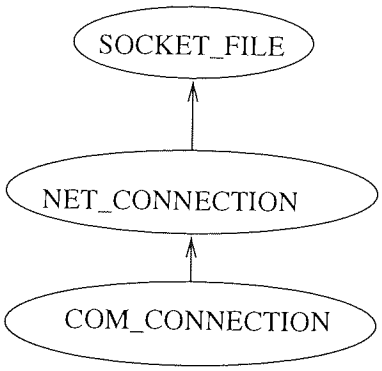


Figure 10.5: Net-connection inheritance hierarchy

NET_CONNECTION: A complete reimplementation of Hillman's (1990) class. It has been rewritten to provide a more file-like abstraction to the ends of a connection. Instances of this class appear at either end of a connection and contain indicators of whether or not the connection is "special" (i.e. reserved for some purpose such as accepting connections) or a generally usable connection. As indicated by the inheritance hierarchy in figure 10.5, it is descended from *SOCKET_FILE*.

SOCKET_FILE: An abstraction over the implementation of sockets, providing various routines: to connect to an object on a specified computer using a specified port number; to accept connections from another object; to indicate the presence or absence of waiting messages; to transmit and receive various items of data including integers, reals, characters, booleans, strings, etc., and of course coded_strings.

COM_CONNECTION: Specialised *NET_CONNECTION* adding features to provide direct support for the types of connection required within the implementation of a COM. It includes the name of the remote object, the name of the owner of the connection,

the specific *OBJECT_PORT* used on the remote object, and send, receive, and reply routines to enable sending of objects across a connection.

10.3 Locking Objects

Chapter 8 implied a need for the ability to lock an object, whether it is COM-based or of simple-type. Further, associated with locking is the idea of early-returns (see section 8.6). This section outlines a viable implementation strategy. It is based around the use of a class *LOCKABLE* which provides the required locking facilities for an object, including the features lock, unlock, pass_lock and identity of the locking object. This is inherited by any classes describing objects that may require locking at run-time.

10.3.1 Locking

Each object that can be locked is required to inherit from the class *LOCKABLE*. This is achieved by making the *COM* class (see section 10.2.1) inherit from the class *LOCKABLE*. This enables a COM-based object to maintain an exclusive lock upon itself. An object requiring to use any object must request a lock of that object, the object in turn on permitting the lock must ensure that no other object is allowed to execute any of the object's methods - this method is called *lock*.

As discussed in chapter 8 and section 9.4 the lock must be passed along the call path, i.e. it must be possible for a method that has a lock upon an object to release that lock to any method it calls that also needs the lock, thus removing its own access and giving preference to methods it calls. This is achieved with the routine called *pass_lock* which takes as parameters the object that is to receive the exclusive access and an indication of the object that currently has it locked. This routine if permitted stacks the identity of the old locking object and sets the owner of the lock to the new object. When a method-object finishes it calls *unlock*; the locked object unstacks the previous locker passing exclusive-access back to that object.

Whilst an object is locked a request may come in to provide exclusive access to another object; this request is queued and serviced only when the object is unlocked by the method-objects currently using it.

10.3.2 Early Return

The implementation of early returns (see section 9.5) relies on the blocking provided above so that all shared entities are locked; indeed, any objects which are the subject of an assignment

involving a query must also be locked. A specific message is used to indicate to the caller that an early return is being performed, so that the caller can take the appropriate action.

10.4 Code Generation

As discussed in chapter 9 the implementation is based upon a "recursive descent"-style process, where, to reiterate, a *do_print* routine is requested of the top construct, which in turn requests a *do_print* action of its component-constructs.

10.5 Actual Execution

The execution framework used to implement the ideas within this thesis makes use of an NFS-based[4] file system. After compilation, the binaries for each of the generated classes are held in a common place within the filing system. The launching of a program is achieved by pulling the appropriate binary from the file-system and starting it. Clearly this does not lead to very high performance levels but it is adequate for demonstrating the thesis.[5]

10.6 End Notes

It should be noted that examples of code generation as discussed in this thesis can be seen in the examples presented in chapter 11 with the associated generated code in appendices C and D. It is however the case that the full locking strategy and its code generation is not yet complete within the implementation; however, it is clear looking at the thesis and the generated results that sufficient semantic information has been obtained from the source program and that there is sufficient infra-structure within the generated code into which such a locking strategy can be inserted.

[4] Network File System

[5] See section 13.3 for a discussion of future improvements in this framework.

Part IV

Evaluation and Conclusions

Introduction to part IV

Chapter 11 evaluates the models presented in parts 2 and 3 - design and implementation. It also relates what has actually been achieved compared with what was originally intended - outlined in chapter 5. This is followed by chapter 12, a discussion of related work (specifically work done during the time of this thesis) and outlines the contributions made in this thesis. This is followed by chapter 13 which discusses how the models and implementation techniques used within this thesis may be refined to achieve a more effective system and the final chapter (14) which presents the conclusion.

Chapter 11

Model Evaluation

This chapter is an evaluation of the models introduced within this thesis; models both for concurrency and for compiler implementation. To discuss the compiler model an example is given which contains some indicative code (see section 11.2) which is discussed and can be compared with the parallel version in appendix B. The compiler model is further exercised in the evaluation of the COM-based concurrency model, where two specific concurrency problems are discussed, the "producer-consumer" problem (see section 11.4) and "dining philosophers" (see section 11.5), both of which can be found in Ben-Ari (1982); these problems were chosen with respect to the evaluation of the concurrency model as, in their parallelised form, they demonstrate the sorts of problems inherent within a parallel system. The compilation model is shown to be demonstrably reasonable by observing the successful translation of the problems above, from their sequential form into their parallelised form; the concurrency model requires further discussion and evaluation and thus is the main focus of this chapter.

With the evaluation of the concurrency model using "producer-consumer" and "dining philosophers" problems it is not immediately obvious how such problems can be simulated in a sequential programming language. The sequential program, which can be parallelised to simulate these problems, must be a valid executable program if the parallelised version is also to be valid. In its sequential form it is clearly not a true simulation of the two problems as they are defined within a parallel domain. Thus the simulation is only manifest in the parallelised version. To reliably achieve such simulations the programs must be written with knowledge of the mechanisms used to achieve parallelisation and subsequent execution.[1]

It is observed in the simulation of the producer-consumer and dining philosopher prob-

[1] This knowledge requirement is not a weakness in either of the models, but arises because of a wish to use the compiler in a way that it is not meant to be used (i.e. trying to control how the system does the parallelisation).

lems that standard "good" object-oriented programming gives rise to a maximised level of parallelism for this COM-based model of concurrency (not necessarily the maximum possible under other models), while poor programming which does not fulfil the principles of section 5.2 will reduce the potential level of parallelism.

It should be noted throughout this chapter that most of the code-generated output from the examples is, due to its lengthiness, incorporated in three appendices (B, C and D) rather than in the body of the chapter. Also, as this output is generated by the code-generator within the compiler, it is not always presented tidily or in an aesthetically appealing fashion.

11.1 Compilation Model

11.1.1 Parsing and Analysis

The parsing phase, implemented using classes within the Eiffel parsing cluster, capitalised upon an object-oriented view placed upon Eiffel's grammar and an input program's structure. The resulting object-based structure of the compiler made it possible to conveniently pass synthesised and inherited (attribute grammar sense) information around; particularly inherited information, to constructs that required information about objects of which they are components, to be able to synthesise their own attributes ready for code-generation.

The construction of the model for compilation and the analysis performed to achieve parallelisation of an input program highlighted the poor way in which many object-oriented programs are written with respect to information hiding and that this has an impact on the degree of potential parallelism. This gave rise to the programming principles of section 5.2.

The successful code-generation of parallelised versions with the "producer-consumer" and "dining philosophers" problem demonstrates that an Eiffel program provides enough semantic information to enable automatic parallelisation. The discussion in sections 11.4 and 11.5 will demonstrate the sufficiency of the synthesised semantic information, such that it is clear that there is at least one automatable path from a sequential Eiffel program to a parallel program.

The demonstration of the correctness of the parallelised code, due to the unfinished nature of the code-generator with respect to the locking algorithms, must of necessity be based on discussion of the theoretical reasonableness of the mapping of parallelised code onto the COM-based model of concurrency; and indeed much of the rest of this chapter looks at this problem.

11.2 Translation Mappings

This section looks at the mapping of constructs from their source to their parallelised version. It includes discussion of if statements, loop statements, method applications, expressions, message handling and the message-passing primitives. These are evaluated in the context of a demonstration class (*CONSTRUCT*) given below and compared with translation extracts in the subsequent subsections.[2] The class's only purpose is as a basis for discussion within this section.

```
class CONSTRUCT export
    meth1, meth2, meth3, n, set_to, cons
feature

    Create is
        local
            i : INTEGER;
        do
            if_statement;
            loop_statement;
            method_application;
        end; -- Create

    n : INTEGER;

    cons : CONSTRUCT;

    if_statement is
        local
            i : INTEGER;
        do
            i := 0;
            if i - meth1.n < 1 then
                i := i + 1;
            elsif i + meth2.n > 2 then
                i := i * 2;
            elsif i * meth3.n = 1 then
                i := i -10;
            else
                i := 1;
            end; -- if
        end; -- if_statement

    loop_statement is
        local
            i : INTEGER;
        do
            from
```

[2] Due to the lengthiness of the generated source code the complete parallelised listings for *CONSTRUCT* are in appendix B with extracts discussed in this section.

```
            i := 1;
        until
            i = meth1.meth2.meth3.n
        loop
            i := i + 1;
        end; -- loop
    end; -- loop_statement

method_application is
    do
        cons.meth1.meth2.meth3.set_to(4);
    end; -- method_application

meth1 : CONSTRUCT is
    do
        Result := cons;
    end; -- meth1

meth2 : CONSTRUCT is
    do
        Result := cons;
    end; -- meth2

meth3 : CONSTRUCT is
    do
        Result := cons;
    end; -- meth3

set_to(i : INTEGER) is
    do
        n := i;
    end; -- set_to

end; -- class CONSTRUCT
```

11.2.1 If Statement

Within the class *CONSTRUCT* the method *if_statement* was included to demonstrate the translation of a general if statement, including *elsif* clauses. It can be seen that, as required, a code-generated version of the condition is pre-evaluated before the actual result of the condition is placed into the resulting if statement's conditional part, and that subsequent "if condition then" parts are nested deeper and deeper inside, with the deepest else clause holding the statements that implement the else part of the source code. This is as expected and as described within section 9.3.2. Further examples of translated if statements can be seen in the "producer-consumer" and "dining philosophers" examples in sections 11.4 and 11.5.

```
local_comp.clear;
```

```
local_msg.clear;
local_msg.set_method_name("meth1");
local_comp.extend(local_msg.coded_string);
local_msg.set_method_name("n");
local_comp.extend(local_msg.coded_string);
if i - master_connection.send_msg(local_comp.coded_string).the_integer < 1 then
    i := i + 1;
else
    local_comp.clear;
    local_msg.clear;
    local_msg.set_method_name("meth2");
    local_comp.extend(local_msg.coded_string);
    local_msg.set_method_name("n");
    local_comp.extend(local_msg.coded_string);
    if i + master_connection.send_msg(local_comp.coded_string).the_integer > 2 then
        i := i * 2;
    else
        local_comp.clear;
        local_msg.clear;
        local_msg.set_method_name("meth3");
        local_comp.extend(local_msg.coded_string);
        local_msg.set_method_name("n");
        local_comp.extend(local_msg.coded_string);
        if i * master_connection.send_msg(local_comp.coded_string).the_integer = 1
            i := i - 10;
        else
            i := 1;
        end; -- if
     end; -- if
end;
```

11.2.2 Loop Statement

Within the class *CONSTRUCT* the method *loop_statement* was included to demonstrate the translation of a general loop statement. It can be seen that, as required, a code-generated version of the condition is pre-evaluated before the actual result of the condition is used in the loop condition, both between the "from" and "loop" keywords and at the end of the loop body, once more before the result is used as a loop condition. This is as expected and as described within section 9.3.3. Further examples of translated loop statements can be seen in the "producer-consumer" and "dining philosophers" examples in sections 11.4 and 11.5.

```
from
    i := 1;
    -- extra assignments to support pre-evaluation of condition
    local_comp.clear;
    local_msg.clear;
    local_msg.set_method_name("meth3");
    local_comp.extend(local_msg.coded_string);
    local_msg.set_method_name("n");
```

```
    local_comp.extend(local_msg.coded_string);
    local_msg.set_method_name("meth2");
    local_comp.extend(local_msg.coded_string);
    local_msg.set_method_name("meth3");
    local_comp.extend(local_msg.coded_string);
    local_msg.set_method_name("n");
    local_comp.extend(local_msg.coded_string);
    local_msg.set_method_name("meth1");
    local_comp.extend(local_msg.coded_string);
    local_msg.set_method_name("meth2");
    local_comp.extend(local_msg.coded_string);
    local_msg.set_method_name("meth3");
    local_comp.extend(local_msg.coded_string);
    local_msg.set_method_name("n");
    local_comp.extend(local_msg.coded_string);

until
    i = master_connection.send_msg(local_comp.coded_string).the_integer
loop
    i := i + 1;;
    -- extra assignments to support pre-evaluation of condition
    local_comp.clear;
    local_msg.clear;
    local_msg.set_method_name("meth3");
    local_comp.extend(local_msg.coded_string);
    local_msg.set_method_name("n");
    local_comp.extend(local_msg.coded_string);
    local_msg.set_method_name("meth2");
    local_comp.extend(local_msg.coded_string);
    local_msg.set_method_name("meth3");
    local_comp.extend(local_msg.coded_string);
    local_msg.set_method_name("n");
    local_comp.extend(local_msg.coded_string);
    local_msg.set_method_name("meth1");
    local_comp.extend(local_msg.coded_string);
    local_msg.set_method_name("meth2");
    local_comp.extend(local_msg.coded_string);
    local_msg.set_method_name("meth3");
    local_comp.extend(local_msg.coded_string);
    local_msg.set_method_name("n");
    local_comp.extend(local_msg.coded_string);

end;    -- loop
```

11.2.3 Method Application

Within the class *CONSTRUCT* the method *method_application* was included to demonstrate the translation of a compound method application. It can be seen that, as required, the message is built up as a compound message of all of the required method applications, then that compound message is forwarded to the appropriate object, in this case *cons*. This demonstrates the sending of a command. A further example can be seen in the compilation

of the loop condition in section 11.2.2 where a query is translated. These translations are as expected and as described within sections 9.3.1 and 9.2.3. There are no examples of compound message applications within the "producer-consumer" and "dining philosophers" examples.

```
local_comp.clear;
local_msg.clear;
local_msg.set_method_name("meth3");
local_comp.extend(local_msg.coded_string);
local_msg.set_method_name("set_to");
local_code.set_integer(4);
local_msg.extend(local_code);
local_comp.extend(local_msg.coded_string);
local_msg.set_method_name("meth2");
local_comp.extend(local_msg.coded_string);
local_msg.set_method_name("meth3");
local_comp.extend(local_msg.coded_string);
local_msg.set_method_name("set_to");
local_code.set_integer(4);
local_msg.extend(local_code);
local_comp.extend(local_msg.coded_string);
local_msg.set_method_name("meth1");
local_comp.extend(local_msg.coded_string);
local_msg.set_method_name("meth2");
local_comp.extend(local_msg.coded_string);
local_msg.set_method_name("meth3");
local_comp.extend(local_msg.coded_string);
local_msg.set_method_name("set_to");
local_code.set_integer(4);
local_msg.extend(local_code);
local_comp.extend(local_msg.coded_string);
local_msg.set_method_name("meth1");
local_comp.extend(local_msg.coded_string);
local_msg.set_method_name("meth2");
local_comp.extend(local_msg.coded_string);
local_msg.set_method_name("meth3");
local_comp.extend(local_msg.coded_string);
local_msg.set_method_name("set_to");
local_code.set_integer(4);
local_msg.extend(local_code);
local_comp.extend(local_msg.coded_string);
cons.send_command(local_comp.coded_string);
```

11.2.4 Expression

Examples of expression translation can be seen with the evaluation of the conditions in the loop (section 11.2.2), and in the if statement (section 11.2.1). In both of these cases method applications are involved. Simpler expression translation can be seen within the body of the aforementioned loop and in the statements to be executed within the if statement.

Further examples of expression translation can be seen in the "producer-consumer" and "dining-philosopher" examples. In all cases the expression translations are as specified within section 9.1.3 with all temporary local variables generated and declared within the appropriate generated methods as required.

11.2.5 Message Handling and Primitives

The primitives are logically what was described in section 7.2, however, during implementation they are also wrapped up inside other calls, which do not change what was stated in section 7.2 but do make code-generation quicker and easier. Specifically a *COM_CONNECTION* exports *send_msg*, *receive_msg* and *reply_msg* which take the message as parameters and forward or receive them along the *COM_CONNECTION*. Also a *NET_CAPABLE* (i.e. any network transmittable message) exports *com_send*, *com_receive* and *com_reply* which fulfil directly the semantics of section 7.2. The *receive-from-anything* semantics are achieved in that a COM listens for any messages and, given a "message detect", calls *receive* on the appropriate connection.

11.2.6 Features

The translation of features as outlined in section 9.1.2 requires that the generated class inherit from the class *COMMAND* and implements a number of routines, *pre_*, *post_* and *do_method*. As can be seen in the appendix for each of the features of *CONSTRUCT* this has been achieved. A further requirement was that the *pre_method* locked any necessary variables based on a system-wide ordering. As stated in section 10.6 the locking is not completely implemented at this stage, therefore it is shown by some code-generation (look at the calls using *set_give_me*) that the *pre_method* "knows" which attributes it needs to ask its controller for, and indeed checks whether a reconnection is required to any COM-based objects because they have moved ports (i.e. the alias in a program which refers to an object now refers to a different object, possibly by assignment). Correspondingly *post_method* is required to release the objects in a system-wide ordering, and again because of the non-completion of locking it has been shown (look at the calls using *set_object_res*) that the method implementation "knows" which objects it must "give back" (i.e. unlock and/or pass any changed attribute values back). The *do_method* routines, as can be seen, are translations (with the expectation of using COM-based semantics) of their corresponding source code.

Methods are made up of either queries or commands; queries return a result and thus if the translation of any of *meth1*, *meth2*, or *meth3* are inspected in appendix B, it can be seen that a reply is generated and sent back to the caller. Also associated with both commands

and queries is the idea of early returns (see section 10.3.2); these can be seen clearly generated in the *pre_methods* of all of the code-generated methods.

Further examples can be seen in the "producer-consumer" and "dining philosophers" problems.

11.2.7 Classes

The largest construct that the compiler is expected to deal with is that of a class. It is anticipated (see section 9.1.1) that a code-generated version of a class will contain definitions for *pre_loop*, *method_connect*, *mk_class_name*, *process_command* and attributes which are COM-connections to each of the code-generated methods. As can be seen by inspection of the parallelised code in appendix B, each of these routines have been generated and a method-class describing how to do the *create* - *MK_CONSTRUCT* - has been code-generated. The *method_connect* contains the necessary code to allow the generated methods to connect to their controller for use, and the *process_command* contains the appropriate code to handle all the method requests that a *CONSTRUCT* might expect, with a forwarding of any other messages to the *BRAIN's process_command* methods such that any COM-system messages or UNIX signals can be appropriately dealt with. Further examples of class parallelisation can be seen in the "producer-consumer" and "dining philosophers" problems.

11.2.8 Inheritance

The section 9.6 outlined the requirement placed upon the inheritance of making a class inherit from the class *COM* with some redefinitions required. This can be seen in the translation of the *CONSTRUCT* class (see appendix B) where the generated class has been made to inherit from *COM*. However, as stated in section 9.6, there is a known problem with the mixing of inheritance and concurrency and therefore none of the example's sources use inheritance; consequently, these programs do not exhibit the aforementioned problem. Obviously as it becomes clearer how such a problem can be resolved, it should be incorporated within the code-generation system of this compiler.

11.3 Concurrent Object Machine

At the heart of the execution environment is an implementation of the Concurrent Object Machine (COM) of chapter 7. The COM was designed to fulfil the purpose of providing a level of abstraction above that of hardware and operating system, upon which the parallelised program must run, and to help simplify the code-generator. It was not designed for optimality but to be a tool for demonstrating the thesis. However, much thought has gone into the

structure of the machine, the message-passing mechanisms and the type of addressing to use (i.e. symmetric or anti-symmetric and whether the naming should be direct or indirect etc.). It has been designed using as "natural" a model as possible such that future refinements can be incorporated that will improve any weaknesses in its optimality.

It should be noted that the message-passing within the models appears high. The models defined achieve much of their behaviour by exchanging messages to effect method execution, request status information or enact locking. There are of course optimisations possible for intra-machine message-passing (e.g. use of shared variables). However, the overhead of inter-machine message-passing may not optimise sufficiently well to make the approach presented within this thesis generally applicable.[3] It is therefore probable that refinements and optimisations will make COM-based solutions usefully applicable on a limited range of problems only.

The COM executes on a network of UNIX workstations using Berkeley sockets and the UNIX *fork* call. It has been exercised by the execution of some very simple programs written to enable communication of information and remote starting of objects on both the same machine and across machine boundaries.

Although not the objective, it is usable by any programmer who wishes to do any explicitly controlled general network programming in Eiffel, and provides a useful abstraction for that task. This usefulness means that the model fulfils the necessary objective placed upon it - its ease of use and ability to work across networks means that the writing of the code-generation code was simplified when compared with code-generating down to the operating system and directly manipulating sockets and forks throughout the code-generated programs.

11.4 Producer-Consumer Problem

11.4.1 The Problem

The producer-consumer problem, or more specifically the bounded-buffer variant as discussed here, is an abstraction of an aspect of applications that arises when one or more processes produce data that must be stored until another process (or processes) is ready to consume it (i.e. utilise the data in some way). Clearly resources are finite so the buffer, where the data is stored, must in practical terms be limited in size (clearly if this bounded buffer version of the problem is demonstrable then so also is the simpler version which uses an infinite buffer).

[3] This problem is discussed further in section 12.2.4

11.4.2 A Sequential Description

One object-oriented solution to the producer-consumer problem would incorporate buffer, producer and consumer objects. The resultant source classes are described below:

BOUND: this class encapsulates the program which pulls together the buffer, producer and consumer objects into a solution.

PRODUCER: this class models the object that produces the data.

CONSUMER: this class models the object that consumes the data.

BUFFER: this class models the object where intermediate results are stored; this must be of a limited size to reflect the "bounded buffer" aspect of the problem.

The solution follows the ideals of information hiding, query/command divisions, sharing and containment (see section 5.2). In designing a solution a question arises: should the producer and consumer objects have direct access to the buffer? It is suggested that the solution should maintain a divorcing of these objects; a producer should not need to know where its results are going, a consumer should not need to know where its input is coming from; the buffer is independent of both the producer and consumer ideas. If the buffer is incorporated into both the producer and consumer classes and then exported, then it is being modelled as a shared object (see section 5.2.1). This however is both unnecessary and poor design as it puts too much dependency between the producer and consumer objects, thus reducing extendability and reusability. Therefore the "glue" which pulls together these three reusable components is the *BOUND* class which provides the algorithmic solution to the problem.

BOUND

The algorithm provided in the class below provides the sequential solution to the producer-consumer problem on a bounded buffer which, when parallelised, will give rise to a simulation of the producer-consumer problem in a COM-based model. In practice the parallelised version does not possess the levels of parallelism of an explicitly parallel solution (see section 11.4.4), but it does give rise to a level of parallelism inherently absent from execution of the sequential version of the program.

```
class BOUND

feature

        b : BUFFER;
```

```
        producer : PRODUCER;

        consumer : CONSUMER;

        Create is
                do
                        producer.Create;
                        consumer.Create;
                        from
                                producer.produce;
                                b.append(producer.lastprod);
                        until
                                False
                        loop
                                if not b.isempty then
                                        consumer.consume(b.item);
                                        b.take;
                                end; -- if
                                if not b.isfull then
                                        producer.produce;
                                        b.append(producer.lastprod);
                                end; -- if
                        end; -- loop
                end; -- Create

end; -- BOUND
```

PRODUCER

The producer object needs at least two features: one to produce a value (a command) and one to return the most recently produced value (a query). This division into two operations, instead of a single *produce* operation which produces and returns a value, maintains a command/query division with side-effect-free queries (see 5.2.3).

```
class PRODUCER export
        produce, lastprod
feature

        produce is
                -- does whatever is required to produce
                do
                        -- generate data and put it into
                        -- lastprod
                end; -- produce

        lastprod : INTEGER;

end; -- class PRODUCER
```

CONSUMER

The consumer object needs at least one feature: a routine to make use of a supplied value.

```
class CONSUMER export
        consume
feature
        consume(v : INTEGER) is
                do
                        -- code to make use of value v
                end; -- consume

end; -- class CONSUME
```

BUFFER

The buffer has been implemented using a number of comments to indicate where certain commands would be placed. These comments appear in the parallelised version of the code[4] at the points where, if they had been statements, their code-generated output would have been placed. [5]

If an item is to be inserted into the buffer, the required pre-condition is that the buffer is not full; before an item is removed from the buffer the required pre-condition is that the buffer is not empty.[6]

As might be expected, none of the internal variables have been exported and the interface has been made independent of the strategy used to implement the buffer. Also the query/command division has been maintained with the separation of the inspection of a value and its removal from the buffer.

```
class BUFFER export
        isempty, isfull, append, item, take
feature
        BUFFERSIZE : INTEGER is 100; -- size of bounded buffer

        in, next_out : INTEGER;
        n : INTEGER; -- number in array

        Create is
                do
                        -- Allocate(0,BUFFERSIZE); -- sets up buffer
```

[4] As may be observed, comments are maintained in the parallelised version, thus aiding both traceability and readability.

[5] This has been done because an array implementation has not been written to fit in with this work, and it is not advisable to mix the standard Eiffel libraries in with the source code which is being parallelised by this compiler, specifically because they will include pre-conditions, generics, etc., which are not supported.

[6] In a standard Eiffel compiler these conditions could be specified with the use of assertions, but this work does not included their implementation.

```
        end; -- Create;

isempty : BOOLEAN is
        do
                Result := n=0;
        end; -- isempty

isfull : BOOLEAN  is
        do
                Result := n=BUFFERSIZE;
        end; -- isfull


put(in,v : INTEGER) is
        do
                -- code to actually put value in the buffer
        end; -- put

append(v : INTEGER) is
        do
                put(in,v);
                in := in + 1;
                if in = BUFFERSIZE then
                        in := 1; -- Eiffel arrays start at 1
                end; -- if
                n := n + 1;
        end; -- append

get(place : INTEGER) : INTEGER is
        do
                -- selects the item at point in the array
        end; -- get

item : INTEGER is
        -- returns the value the next_out item
        do
                Result := get(next_out);
        end; -- item

take is
        -- remove next_out item
        do
                next_out := next_out + 1;
                if next_out = BUFFERSIZE + 1 then
                        next_out := 1;
                end; -- if
                n := n - 1;
        end; -- take


end; -- BUFFER
```

11.4.3 The Parallelised Version

The parallelised version of the above sequential description should give rise to method-objects for each of the methods, message handlers in each of the objects which have been turned into COMs (i.e. the instance of *BUFFER* and of *BOUND*) and all code modified to fit within this new environment. This is the case, as can be observed by inspecting the source code above and the corresponding code-generated classes in appendix C. The parallelised version has been placed in the appendix due to its length.

11.4.4 Evaluation of Potential Parallelism

Analysis of the code outlined in the *BOUND* class may suggest that it is a little contrived to have wrapped the consuming part with a test for buffer emptiness and the producer part with a test for buffer fullness, even given the stated pre-conditions (see section 11.4.2) for extraction or insertion of values. A programmer writing a sequential program could justifiably avoid such tests as it is clear that the buffer's insertion and extraction of values alternates within the program. In practice, whether or not the conditions are included, the parallelised code executes in a similar fashion; the resulting parallelised code will typically only alternate the consuming of a single value and production of the next value. This alternation within the parallelised program is because of the requests for locking upon the buffer and the need to maintain the semantics expected from executing the program sequentially.

Looking in more detail at locking, the body of class *BOUND* (see the *Create* routine) contains three objects that must be locked at different points: the *consumer*, *producer* and *buffer*. The methods applied to each require access to only one object at a time e.g. buffer's methods *append*, *isempty* and *isfull* only require access to the buffer, the *append* routine which is parameterised is not called until the result of *producer.lastprod* [7] has been evaluated. Thus one of the requirements for deadlock is absent. None of the methods in this example require more than one object to be locked and the calling pattern does not in this case form a circle of methods requesting each other and requiring access to another method's locked objects.[8] The lock alternates between the application of various of the buffer's exported routines applied within the *BOUND* class e.g. the *append* routine gains the buffer from the *BOUND* object; during this time the *BOUND* class continues to execute (assuming an early return from the *append* routine); thus it will then apply the *isempty* routine, because the COM-based strategy only permits one external method request to be serviced at a time and

[7] *producer.lastprod* in turn is not executed until *producer.produce* has completed, for two reasons: *producer's lastprod* will have been locked by the producer's *produce* method; and more significantly COM-based objects can only deal with one external method request at a time.

[8] Given the technique for locking if a method $meth_1$ called a $meth_2$ which required access to an object locked by method $meth_1$ then $meth_2$ would have obtained the lock from $meth_1$ (see section 8.8.4).

indeed if *append* is currently executing then the buffer is locked by one of its own routines; the code in the *BOUND* class must await the return of the lock. Having gained the lock once more it can check for emptiness and proceed. If the buffer is not empty it will consume a value by asking the buffer for the next *item*. Having gained the item the consumer object can proceed (again presuming an early return from *consume*); meanwhile the lock of the buffer will once more be with the *BOUND* object and thus it can apply *b.take* and so on. As can be seen the lock of the buffer will pass from the *BOUND* class to the buffer itself as it applies one of its own methods and back, suspending execution in the *BOUND* object. The consumer and producer object, having gained the information they need, can proceed with consuming or producing in parallel (as they require no access to the buffer). Thus we have the parallelism achieved by interleaving the *produce* and *consume* methods but because of the encoded locking strategy the producer cannot proceed ahead of the consumer producing say ten items whilst the consumer manages to consume say two or three.

There is, therefore, with this solution, an increase in the potential parallelism over the sequential program, but it does not reach the levels that might be expected from a more typical solution to the producer-consumer problem; in fact a large buffer is redundant in that there can be at most one extra value awaiting consumption. This highlights an area of potential future work: extend the current locking strategy. The current approach means that alternating method calls, when sharing an object, if repeated cannot progress far beyond each other. To avoid this would require a different locking technique such that a common object (e.g. a buffer) could be shared but because the producer and consumer operations are referring to different parts they could be parallelised. This was proposed as an area of future work in section 8.9, such that an object could handle multiple external requests simultaneously; this assumes that it can be decided practically that the two methods do not interfere with each other, and that the original program's sequential semantics are maintained.

11.5 Dining Philosophers

11.5.1 The Problem

The dining philosopher problem demonstrates in a vividly graphical situation the pitfalls of concurrent programming research (Ben-Ari 1982).

> The problem is set in a monastery whose five monks are dedicated philosophers. Each philosopher would be happy to engage only in thinking were it not occasionally necessary to eat. Thus the life of a philosopher is an endless cycle: **repeat** *think*; *eat* **forever**.

Figure 11.1: Dining Philosophers

> The communal dining arrangement is shown in Fig.11.1.[9] In the center of the table is a bowl of spaghetti that is endlessly replenished; there are five plates and five forks. A philosopher wishing to eat enters the dining room, takes a seat, eats and then returns to his cell to think. However, the spaghetti is so hopelessly entangled that the two forks are needed simultaneously in order to eat. ...
>
> The problem is to devise a ritual (protocol) that will allow the philosophers to eat. Each philosopher may use only the two forks adjacent to his plate. The protocol must satisfy the usual requirements: mutual exclusion (no two philosophers try to use the same fork simultaneously) and freedom from deadlock and lockout (absence of starvation - literally!) An additional safety property is that if a philosopher is eating then he actually has two forks. (Ben-Ari 1982, pp 109-110)

11.5.2 A Sequential Description

One object-oriented solution to the dining philosopher problem would incorporate fork and philosopher objects. The resultant source classes are described below:

DINING: this class encapsulates the program which pulls together the philosophers and forks into a solution.

PHILOSOPHER: this class models a philosopher.

FORK: this class models a fork.

The solution, as with that for the producer-consumer problem of section 11.4, maintains the ideals of information hiding, query/command divisions, sharing and containment (see

[9] Figure number modified to fit in with thesis figure numbering

section 5.2). This problem, however, is a little more awkward to *simulate* using sequential code and object-oriented ideas. It is normal for the philosopher, in solutions to this problem, to have a "repeat-forever" loop as outlined in the earlier quote. However, such an infinite loop placed into a sequential program would mean that the first philosopher that was told to "live" i.e. *think* and *eat*, would never return control to the caller. Therefore to try to get some way towards a solution that will map to a parallel form of the dining philosophers it is necessary to avoid such loops. Instead within the class *DINING* there is a control loop which, based on whether or not a philosopher is hungry, tells the philosopher to eat or think appropriately. Clearly the sequential program given below cannot suffer from deadlock when executed sequentially but more importantly neither does the parallelised program; also, the parallelised program does not suffer from either deadlock or starvation but does have a greater level of potential parallelism (see section 11.5.4).

DINING

The algorithm provided in the class below provides a sequential *simulation* of the dining philosopher problem. Clearly when running sequentially there is no problem of contention for forks, etc., as only one philosopher can be thinking or eating at a time. However when this code is parallelised there arises the potential for deadlock and starvation which must be (and is) avoided by the concurrency model presented within this thesis.

```
class DINING

feature

        phil1, phil2, phil3, phil4, phil5 : PHILOSOPHER;

        f1,f2,f3,f4,f5 : FORK;

        Create is
                do
                        phil1.Create;
                        phil2.Create;
                        phil3.Create;
                        phil4.Create;
                        phil5.Create;
                        f1.Create;
                        f2.Create;
                        f3.Create;
                        f4.Create;
                        f5.Create;
                        live;
                end; -- Create
```

```
        live is
                do
                        from
                        until
                                False
                        loop
                                if phil1.hungry then
                                        phil1.eat(f1,f2);
                                else
                                        phil1.think;
                                end; -- if
                                if phil2.hungry then
                                        phil2.eat(f2,f3);
                                else
                                        phil2.think;
                                end; -- if
                                if phil3.hungry then
                                        phil3.eat(f3,f4);
                                else
                                        phil3.think;
                                end; -- if
                                if phil4.hungry then
                                        phil4.eat(f4,f5);
                                else
                                        phil4.think;
                                end; -- if
                                if phil5.hungry then
                                        phil5.eat(f5,f1);
                                else
                                        phil5.think;
                                end; -- if
                        end; -- loop
                end; -- live
end; -- class DINING
```

FORK

The *FORK* class below exports three routines, one to pick a fork up for use, one to use the fork, and another to put the fork down, called *pickup*, *use*, and *putdown* respectively.

```
class FORK export
        pickup, putdown, use
feature
        pickup is
                do
                        -- code to pickup fork
                end; -- pickup

        use is
                do
                        -- code to actually use the fork
                end; -- use
```

```
        putdown is
                do
                        -- code to putdown fork
                end; -- putdown


end; -- class FORK
```

PHILOSOPHER

As was discussed in section 11.5.2, infinite loops cannot be used in sequential programs where control must be returned from a routine containing an infinite loop. This means that they cannot be used in the source code of the programs being parallelised for the same reason, as the parallelised version tries to produce a semantically equivalent simulation, hence the effect of an infinite loop in a sequential program will also be seen within the parallelised version; specifically, it will permanently lock shared variables. Consequently the *eat* and *think* methods which a philosopher is expected to perform are separated into two routines, called by the control part in class *DINING* based upon the *PHILOSOPHER* exported feature *hungry* which indicates when a philosopher wishes to eat.

```
class PHILOSOPHER export
        eat, think, hungry
feature
        hungry : BOOLEAN;

        eat(f1,f2 : FORK) is
                local
                        ate : INTEGER;
                do
                        f1.pickup;
                        f2.pickup;
                        from
                        until
                                not hungry
                        loop
                                -- do some eating using forks
                                f1.use;
                                f2.use;

                                ate := ate + 1;
                                -- set hungry to False when full, e.g.
                                if ate > 10000 then
                                        hungry := False;
                                end; -- if
                        end; -- loop
                        f2.putdown;
                        f1.putdown;
```

```
                end; -- eat

        think is
                local
                        thunk : INTEGER;
                do
                        from
                        until
                                hungry
                        loop
                                -- do some thinking
                                thunk := thunk + 1;
                                -- set hungry to True when hungry, e.g.
                                if thunk > 10000 then
                                        hungry := True;
                                end; -- if
                        end; -- loop
                end; -- think

end -- class PHILOSOPHER
```

11.5.3 The Parallelised Version

The parallelised version of the above sequential description should give rise to method-objects for each of the methods, message handlers in each of the objects which have been turned into COMs (i.e. the instance of *BUFFER* and of *BOUND*) and all code modified to fit within this new environment. As with the producer-consumer problem, all of the expected classes and translations were performed. The parallelised produced code is again quite long, therefore the full translation is placed in appendix D.

11.5.4 Evaluation of Potential Parallelism

The solution outlined for simulating the dining philosopher problem using the model of concurrency presented within this thesis gives rise to a high level of parallelism. As discussed in section 11.5.3, the source classes led to the expected classes.

The philosophers and forks are all created within *DINING* and then the routine *live* is called. The infinite loop within *live* then tests whether each philosopher is hungry; if any are, the algorithm in *DINING* sends the message *eat*, with the appropriate forks as parameters; any philosophers that are not hungry it instructs to *think*.

The solution enables total parallelism between all of the philosophers if they are thinking; as there are no common objects between these routines, there is nothing that prevents the progress of the *think* methods. If only one of the philosophers becomes hungry then total parallelism is still possible, as the *eat* of the one philosopher will not have any conflicts over forks with any other philosopher. It is when more than one philosopher requires to eat that

issues of deadlock and starvation (both literally and in the deadlock sense) arise.

It can be seen that because of the requirement that objects must be locked in an order based on a system-wide ordering of objects (see section 8.8.3 for the reasoning) that no deadlock will occur, and because the activities are finite that starvation will also be avoided.

Consider for example $phil_1$[10] has $fork_1$ and $fork_2$. Philosopher $phil_2$ wishes to eat, however, as $phil_1$ has $fork_2$ locked, $phil_2$'s request will be queued within $fork_2$ and $phil_2$ will be forced to wait. If meantime $phil_3$ decides that he wishes to eat then he will lock $fork_3$, as it must be free because $phil_2$ is trying to gain a lock on the fork which comes earlier in the system wide ordering of forks. Now $phil_3$ can request $fork_4$ which he may or may not immediately get depending on whether $phil_4$ is eating. Assuming that $phil_3$ does get $fork_4$, the current situation is that $phil_1$ and $phil_3$ are eating thus using both of $phil_2$'s forks. The philosopher $phil_2$ has a request queued for $fork_2$ and consequently will not starve because ultimately $phil_1$ will stop eating, as eating is a finite activity, and thus $phil_2$ will gain $fork_2$ then he will request $fork_3$ and regardless of whether $phil_3$ currently has it locked will eventually gain $fork_3$ because of the finite nature of eating. Thus it can be seen that a philosopher does not starve. It can also be seen that the gaining of forks is as fair as first come first served mechanisms can be.

Consider a different scenario: all of the philosophers decide at the same instant to eat. As they must lock their forks based on a system-wide ordering of objects the following worst case scenario may occur:

- $phil_1$ asks for and locks $fork_1$
- $phil_2$ asks for and locks $fork_2$
- $phil_3$ asks for and locks $fork_3$
- $phil_4$ asks for and locks $fork_4$
- $phil_5$ asks for $fork_1$

As can be seen $phil_5$ will try to lock $fork_1$ first and not $fork_5$ as $fork_1$ comes before $fork_5$ in the ordering of objects. At this point $phil_1$, $phil_2$ and $phil_3$ have their first forks but cannot proceed; their requests for their other required fork will cause them to be placed on a queue for the use of the appropriate fork and they will block. Neither can $phil_5$ proceed as his first required fork is locked by $phil_1$ and he has been placed in a queue to use that fork. However $phil_4$ can proceed, his next required fork, $fork_5$, is free as $phil_5$ will not have tried

[10] $phil_n$ and $fork_n$ are used as specified in the *DINING* class, also it is assumed that the system-wide ordering makes $fork_n$ come after $fork_{(n-1)}$

to lock it yet, again because of the ordering on objects. So $phil_4$ eats, and because of the finite nature of eating he will ultimately release both of his forks, this will mean $phil_3$ can *eat* and $phil_4$ can *think*; all other philosophers are still blocked waiting. This will proceed backwards through the fork ordering, releasing a further philosopher to eat, and another to think, until ultimately $phil_5$ gets to eat.

Consider however, at the point in this scenario when $phil_2$ starts to eat (say), $phil_4$ may be a very hungry philosopher and want to eat again; as $phil_5$ is still locked out and assuming $phil_3$ is still thinking then $phil_4$ can eat again immediately causing $phil_5$ to wait even longer even if $phil_1$ finally finishes eating thus releasing $phil_5$'s first fork. However $phil_5$ cannot be made to starve as $phil_4$ will ultimately finish, but he might not be as quick to eat as he may have been if he had had the ability to lock $fork_5$ as soon as $phil_4$ had finished.

This is the worst possible case scenario and yet it can be seen that the model is devoid of both deadlock and starvation and is reasonably fair in that even with $phil_4$'s big appetite $phil_5$ will not be locked out indefinitely. This model is similar to standard solutions presented by Ben-Ari (1982) with respect to deadlock, starvation and fairness.

11.6 Summary

It has been shown that the model used to implement the compiler derives sufficient semantic information from the sequential Eiffel program such that an increase in potential parallelism is gained over that of the sequentially executed program. It has also been argued that the use of a good programming style, as discussed in section 5.2, which simply encourages the use of sound software development practice, leads to a greater level of potential parallelism in the COM-based model of concurrency described within this thesis. It has also been shown that the compiler, given the derived semantic information, produces all of the appropriate classes with the attributes, expressions, messages, methods and classes appropriately generated for a COM-based implementation.

It has been further argued that it is possible to have a locking system that enables reliable locking which avoids deadlock and starvation and also provides the necessary support for mutual exclusion. The construction of the compiler demonstrated that the COM does provide an effective and useful abstraction layer for parallel programming on a distributed system, at least as the target of a code-generator.

Chapter 12

Related Work and Contributions

This chapter outlines related work (specifically work that has taken place during the later part of the work on this thesis) and summarises the contributions made by this thesis. The areas discussed include compiler implementation, parallelising Eiffel and inheritance.

12.1 Compiler Implementation

The use of object-oriented techniques for writing parsers was at the start of this work an area which was beginning to be investigated. There was Meyer & Nerson's (1990*b*) work with their construction of classes to help write scanners and parsers. This, as stated in the thesis, has formed the base for much of the compiler-writing work. Also around the time that this work was started (i.e. September 1989) there was a preliminary paper by Hucklesy & Meyer (1989) which discussed an extension to the Eiffel parsing libraries in very general terms which would provide a tool similar to the yacc parser generator for generating object-oriented parsers. This idea of a parser generator has within recent years been substantially extended: the work of Avotins, Mingins & Schmidt (1995) in presenting YOOCC (Yes! an Object-Oriented Compiler Compiler) has achieved some level of maturity, providing a tool which generates Eiffel code describing a parser from grammar descriptions. However, these works, at present, concentrate on trying to present an object-oriented parser; they do not provide the same level of support for semantic analysis and subsequent code-generation.

Other work that has been going on, once more during the work for this thesis, is that of a compiler-compiler tool called Cocktail (Grosch & Emmelmann 1990); this tool provides a complete compiler-compiler tool for dealing with all aspects of compiler generation including dealing with semantic analysis and code-generation. It can be used to describe and compile object-oriented programming languages; it is however not itself object-oriented. Apart from

the lack of general availability until 1992 (see ftp://ftp.karlsruhe.gmd.de/pub/cocktail) for both documentation and the tool, it was not suitable for the level of control needed in order to write the compiler for this work. The table-based approach of LR parsers, with the need to formalise the grammar and semantic analysis etc., would not have fitted in well with the need to gradually develop phases and experimentally work out if there was sufficient information.

So to summarise this thesis's contribution in this area: an object-oriented discipline is applied to the process of compiler construction, using recursive descent processing; this of itself has been done in the work above, particularly with YOOCC. However, the demonstration of how to incorporate an attribute grammar style approach with synthesis and inheritance of attribute values within the context of an object-oriented recursive-descent processor is, as far as the author is aware, new. The published work discussed above, with YOOCC and the Eiffel parse libraries, focuses on scanning and parsing and says little about semantic analysis and code generation.

Also whilst writing this compiler a number of things were noticed about Eiffel. There is an apparent semantic flaw within Eiffel, giving rise to the fact that attributes which are exported from a class can be altered even though they are supposedly made available on a read-only basis. It has been demonstrated, through the use of the semantic analysis techniques used within this thesis, that it is in practice possible to detect and either warn about or disallow the use of such a semantic action. The required analysis is not too time-consuming when compared with the benefits of detecting such an errant programming style.

Further, it was demonstrated that with the application of simple principles (suggested in section 5.2) with a "good object-oriented software development" practice - maximising an appropriate use of containment and sharing, with a rigid distinction between queries and commands - that the potential for automatically derived concurrency is increased. Also the consequent cost and complexity of the deadlocking strategy can be minimised.

12.2 Parallelising Eiffel

During the development of this thesis a number of papers have been published detailing approaches to the parallelisation of Eiffel (Gunaseelan & LeBlanc, Jr 1992, Meyer 1993, Wolff 1995, Caromel 1989, Caromel 1990, Jezequel 1993, etc.). Most of the work published, however, specifically the works on Distributed Eiffel by Gunaseelan & LeBlanc, Jr (1992) and the works by Caromel and Meyer (1993) do not address the thesis discussed within this document. The works with EPEE (see section 12.2.1) and Heron (see section 12.2.2) do however fall within the scope of the thesis. Also, although Meyer's (1993) paper does not directly relate to the thesis, it does have implications on how Eiffel may move, in the form

of a standard, towards parallelism given the author's original presentation of the language. EPEE, Heron and Meyer's (1993) paper are discussed below.

12.2.1 The Eiffel Parallel Execution Environment (EPEE)

EPEE is an approach to achieving parallelism for the "normal" programmer whilst hiding the intricacies inherent in the problem. It succeeds for a limited set of problems, but fails for general object-oriented programming. It appears to be an approach that gives an excellent vehicle for implementing matrix style manipulations but it does not appear that it would be quite so effective on a program that did not involve aggregate data structures containing information of the same type.

EPEE provides a set of methods to access, redistribute, and perform methods on elements of parallelised-aggregate-data structures. The suggested development method appears to require two levels of programmer. The first is the library developer who constructs the parallelised versions of standard classes, such as arrays, sets, lists, trees, etc., providing a sequential-style interface. The second programmer is the "normal" programmer who writes the application program that utilises the parallel classes.

This seems a reasonable approach if the applications programmer is able to utilise the standard data-structures without needing to make small changes to them via inheritance to fit in with their application. It would appear from the presentation in one of the EPEE papers (Jezequel 1993) that because of the way in which the parallelised classes are implemented a client usage (i.e. attributes using an EPEE type) in a program is okay, but an inheritance based usage would imply a need to know about the parallelism mechanisms used or alternatively the programmer would have to adhere to some very stringent restrictions on what they could change, if unpredictable crashes and conflicts between inheritance and synchronization were to be avoided.

Given a restricted approach to the use of the parallel classes this method seems to offer a very useful mechanism to achieve a good level of parallelism on problems that one may typically have written in FORTRAN and expected a parallelising compiler to optimise (i.e. a data parallel type problem such as array or matrix manipulations). Jezequel's (1993) paper demonstrates a level of scalability beyond that achieved by parallelising FORTRAN compilers (which it is claimed very rarely scale past 10 processors); the graphs in the paper demonstrate scalability up to 32 processors. It does not, however, appear to be a general solution; its applicability to a general object-oriented program seems minimal. Clearly, therefore, EPEE does not resolve the main thesis of this work. The normal programmer cannot use the full object-oriented philosophy (re: inheritance) without knowledge of the parallelising mechanisms involved.

Two interesting conclusions from this paper were that, because of the underlying models (my opinion), dynamic binding was expensive and exception handling did not emulate the normal sequential behaviour. Neither of these are problems within the models presented within this thesis as, like ISE (Interactive Software Engineering) compilers, dynamic binding is achieved by a parallel equivalent of a direct procedure call and the models maintain the original program's call paths and thus exceptions are propagated and dealt with as expected in the sequential program.

It is clear, however, that the models used to implement EPEE do achieve a greater level of parallelism than the models presented within this thesis (at least while the COM-models in the thesis lack the refinements suggest in chapter 13) and indeed their models have direct support for distribution.

In summary, the EPEE approach should achieve greater performance levels on data parallel problems but the approach suffers from not being completely compatible with the expected object-oriented development approach (at least in Eiffel), inherently restricting how the parallel classes can be used and making safe extensions via inheritance difficult.

12.2.2 Heron

Heron, described as "Transparent heterogenous distribution of high-level object-oriented programs" (Wolff 1995) is an approach to parallelising object-oriented programs in Eiffel, which has very similar objectives to those within this thesis. They try to maintain the syntax and semantics of Eiffel once parallelised and also reuse the original inheritance structure. Wolff (1995) outlines the system design and implementation used in the Heron project. Unlike the work within this thesis, they are working in the later Eiffel v3 as opposed to Eiffel v2.

The Heron approach is transformational in nature, as in this thesis. However, the implementation takes a different perspective: the work in this thesis uses the COM-model and constructs all parallelised objects around instances of it; the approach in the Heron project is to provide a manager upon each machine which manages all of their generated objects and the communications between them. This thesis's approach has totally self-contained objects that control their own behaviour, protection and locking; the Heron approach has objects that are controlled by an external run-time environment. The difference in approach possibly arises because of the different perspectives on the problem. Inspecting the references attached to Wolff's (1995) paper, he has spent a number of years looking at the problems of programming and managing distributed systems. This has arguably led to a raising of the object management in the modelling to a level above the objects; the objects are treated as things to be controlled. My work in comparison, with respect to this thesis, has been focused on object-oriented modelling and compiler construction and consequently (perceiving the

objects as entities that should manage themselves) placed all control within the appropriate objects. Both models ultimately achieve the same objectives and are equally valid. However, Wolf's model may ultimately be more difficult to reason formally about if the approach of "Design by Contract" (Meyer 1988) can be applied to object-oriented parallelism (see section 12.2.3), and indeed it is arguably less "natural" a model. Wolff (1995), however, has been able to utilise PVM for the underlying communication mechanism rather than having to implement his own; thus any improvements in PVM will inherently realise improvements in his implementation.

The biggest problem with Heron is that it is unclear from Wolff's (1995) paper whether Heron deals with the management of deadlock or any other parallel system properties, and it is equally unclear how these could be cleanly incorporated into his model. Further, his inheritance, as with EPEE above, is dealt with purely at run-time and is not optimised into a single call; this leads to inherent inefficiency in a distributed environment. As with this work (see chapter 13), Heron does not properly support "once" operations yet, something Wolff's (1995) paper describes as Eiffel's 'hidden module'.

12.2.3 Bertrand Meyer's Work

A discussion of related work would not be complete without discussion of contributions made by B. Meyer within the period of this thesis, specifically as he is the originator of Eiffel. There is an ongoing discussion of how to put explicit parallelism into Eiffel; how with minimal changes (one extra keyword, and some semantic modifications) it should be possible to come up with a concurrent Eiffel. This would not solve the problems presented in the thesis, as the discussed mechanisms encourage explicit parallel management, but there are a number of related issues that arise.

Meyer's (1993) paper looks (amongst other things) at the use of assertions, specifically within the domain of a concurrent programming language. It looks at the notion of "Design by Contract" (Meyer 1988) and the benefits that accrue from the use of the assertion mechanisms. He claims that this idea of "Design by Contract" provides the starting point for a potential formal approach to object-oriented computation, assuming that proof rules were available for the inner details of an object-oriented programming language. The paper goes on to discuss the need to try to preserve the command-query distinction within concurrent programming and indeed produces a *BOUNDED_QUEUE* (similar in style to the one used in this thesis (see section 11.4)) which separates the *item* and *remove* routines. He goes on to claim that this distinction will in fact suggest some of the important properties of the concurrent mechanism, as indeed it has for the COM-model discussed in this thesis. From here the paper discusses support for many different forms of programmable concurrency (shared

memory, multitasking, network programming, distributed processing, real-time applications) and highlights that a language mechanism cannot provide all of the answers for such a wide spread of application, but should be adaptable to all the intended forms of concurrency (this is the main lack in the EPEE approach discussed previously).

The paper, as with EPEE and Heron above, avoids dealing with *"how the sought mechanism will help solve a difficult problem of concurrent programming: avoiding deadlock"* (Meyer 1993). It is even stated that *"A solution which guarantees deadlock avoidance in all cases would probably be too limited"* (Meyer 1993).

This is unfortunate, even though justifiable in Meyer's work which does not claim to provide a simple mechanism enabling productivity inprovements on parallel hardware. However a deadlock avoidance mechanism, if designed in at the start as one of the main focuses of the parallelism mechanisms, could gain most (if not all) of the benefits of parallelism without a later programmer having to be concerned about deadlock, which is probably one of the larger problems in concurrency along with properties relating to starvation and the fairness of processes.

12.2.4 Summary

It is the lack of dealing with deadlock that appears to be the major weakness of all three approaches discussed above. Incorporating deadlock avoidance mechanisms into the models after having produced the mechanisms for achieving parallelism and communication will quite probably be awkward and possibly lead to a need to redefine the mechanisms as currently described.

With the exception of Meyer's (1993) paper, the total absence of the topic of properties of parallel systems (deadlock, etc.) within the papers that have appeared may not indicate that the topic has not been dealt with. However, for the presented mechanisms of parallelism to be really effective, particularly for the general programmer who knows little of parallelism and how to program it, they need to ensure a limited proneness to error in use; equally the mechanisms should automatically deal with issues such as deadlock, starvation and fairness. Mechanisms that need to utilise excessive resources in the detection and removal, or avoidance of deadlock should be avoided. The mechanisms should be such that absence of starvation and fairness are inherent. Without these properties the described approaches to parallelism will be effective and fast, on the appropriate problems, but they will each only be really useful in the hands of "good" programmers who can deal with the extra complexities of handling the deadlock mechanisms, or in the case of Meyer (1993) all of the reasoning involved in concurrency.

The contribution of this thesis in the area of parallelising programs written in an object-

oriented programming language such as Eiffel is mainly in the demonstration that an effective and sufficient theoretical model of concurrency can enable essential properties of parallel programs to be achieved, such as deadlock avoidance, absence of starvation and fairness of process in the domain of a consequently practical parallelising object-oriented compiler. The technique described which ensures the absence of deadlock relies upon resources being allocated a unique number which is then used as the basis for a linear locking strategy (Havender 1968). The unique number is the enumeration of an object's position in computational space (i.e. machine address and port on which connections can be made to an object).

As was highlighted in section 11.3, there appears to be a weakness in the models which form the basis for this thesis with respect to their general applicability. In the application of the object-oriented approach to the models, a large amount of message-passing has been specified. This is evident throughout as objects make connections to each other to execute a method, request status or enact locking. It is however the fundamental nature of object-oriented programs that causes most concern with respect to any anticipated performance gains: the "normal" method size is small thus the ratio of message packing and delivery to time for a method's execution is higher than it should be to be generally effective. As discussed earlier in this thesis, the very nature of a parallel program executing on a distributed architecture is such that it is necessary (if performance is to be improved over a sequential solution) for the granularity of the processes to be small and the inter-process communications low. This implies in this system that a method should execute for a long period of time compared to the time it takes to deliver a message. This would probably not be true of most "well-written" object-oriented programs. Thus even with optimisations of these models, it is unlikely that a COM-based solution would be generally effective. Optimised models would only prove effective when dealing with programs that include methods which are in some sense "heavyweight" - i.e. they, along with any intra-object calls on the same machine, would take a long period of time when compared with the time to package up and deliver the original message containing the request to execute the method. The structure of this compiler is such a program.

12.3 Inheritance

One aspect that has developed during the writing of this thesis is the "understanding" of inheritance. As was pointed out in section 9.6, there appears to be a conflict between inheritance and concurrency specifically within the area of synchronization and its redefinition (Matsuoka et al. 1993). However, according to McHale (1994), Matsuoka et al.'s (1993) assumptions may be incorrect; there may indeed be a fundamental problem with inheritance

and the best that can be achieved is an alleviation of the symptoms rather than a solving of the problem.

This problem and its study has been developing throughout the period of this thesis, and is clearly related, but its solution was felt to be beyond the scope of the work undertaken here.

Chapter 13

Future work

We shall not cease from exploration
And the end of all our exploring
Will be to arrive where we started
And know the place for the first time
T. S. Eliot, Four Quartets "Little Gidding"

There is inherent in the process of science that, having finished, one finally knows how to begin. As captured eloquently by Eliot above, having got to the stage of presenting theoretical models, having implemented those models, and finally having evaluated those models, it is only now that one realises how it could have been done differently.

This chapter dicusses those aspects within this thesis that, with the gift of hindsight - or with the use of ideas and tools developed during the period of the thesis - might now be done differently, or extended.

The first, and probably most obvious, piece of future work is to complete the locking implementation; however, this chapter focuses more on what could/should be changed to attain better results. The chapter is broken up into four sections, which suggest possible ways to improve the theoretical elegancies or implementation of the COM model, compiler model, and execution model. The final section considers any other future-work issues.

13.1 COM Model

Throughout the thesis there has been discussion about various ideas which should/could be the basis for future work; those which relate to concurrency are refinements of the original COM.

Relax reductionism: It was stated in section 8.9 that it should be possible to relax this thesis's application of reductionism, with its rigid boundaries, within the COM implementation, enabling more efficient execution and minimising some message passing made necessary because of the boundaries. This should be done without sacrificing too much the theoretical elegancies of the model or indeed its safety with respect to protecting the internals of objects.

Support multiple external messages: The COM model as described took a relatively simple approach to dealing with locking and consequent upon this was that an object could only deal with one external message at a time. Whilst this enabled a level of internal concurrency it did not give rise to the levels that could be achieved if non-interfering external method calls to an object could proceed concurrently.

Internal Locking: Consequent on the above support for executing multiple external messages concurrently is a need to refine the locking strategy. This strategy should enable components of an object to be locked, instead of the whole object, but the strategy must still be demonstrated to avoid deadlock. The need for this was noticed specifically with the "producer-consumer" problem where the bounded buffer used to hold intermediate values was redundant because of the lack of a facility to discriminate in the locking process (see section 11.4.4).

Readers/Writers support: Given the discussion of supporting multiple external messages and a more precise internal locking, there could also be an improved level of concurrency within a parallelised system if there was support for a readers/writers approach to locking.

However, with each of these extensions it is essential that a program observe the same semantics as might be expected when sequentially executing the program that has been parallelised, if the thesis's objective is to be achieved.

13.2 Compiler Model

There have been incorporated within the compiler's implementation various assumptions which simplified the problem being tackled (particularly the "Draconian" limitations of section 9.2.3). Given sufficient time, these assumptions would not be necessary to the models; if sufficient work could be completed on the code-generation phase and some of the encoded communication mechanisms, it would be possible to drop the assumptions altogether.

Relax the "Draconian" approach: The only objects that were permitted to be passed around were COM-based or of a simple value type (see section 9.2.3. This is not

an essential limitation, but its removal does imply a more complex code-generation phase. Associated with this restriction, the communication implementation model used an idea which was termed a *CODED_STRING*; this was an encoding of any network-transmittable messages which were inherently kept as simple as possible. The *CODED_STRING* class has, however, now become redundant due to recent work done by Interactive Software Engineering in producing an *EIFFEL_NET* cluster. These facilities for Eiffel enable the packaging up and forwarding of any Eiffel object across a network. Thus, given sufficient time to extend the code-generator and the use of this *EIFFEL_NET* cluster the "Draconian" approach can be relaxed, with sequential non-simple objects treated in a logically similar way to the simple typed objects within this thesis.

Support Once Operations: At present the compiler does not support *once* style operations. Their support would require the implementation of a "class-object" to hold information on whether a *once* routine had been executed. A "class-object" would be similar to the ideas used to implement method-objects, except they would only contain information about *once* routines and any values associated with *once* queries. Clearly this would need investigating carefully to ensure a good fit with the COM-model.

Implement Genericity: This is a simple, if time consuming, extension to the compiler and does not greatly increase the complexity of the compilation task.

Include Assertion Support: Eiffel has support for assertions in the form of pre-conditions, post-conditions, loop and class invariants etc.; it would be a useful extension to the approach to incorporate these within the models and the implementation. Their support would fit in well with the models used within the thesis, as the current locking strategies would work effectively upon them, and their incorporation would mean that a code-generated program would have all of their benefits as found in a standard Eiffel compiler. Specifically, exceptions would be automatically raised because of an inappropriately called routine where a pre-condition was broken, and debugging would be simplified. This would enable the full support of "Design by Contract" that is associated with Eiffel programming (Meyer 1988).

An Eiffel Library: For this compiler to become really useful it would be necessary to implement a concurrent Eiffel library, which provides all of the standard Eiffel clusters. It is not sufficient to try to use this compiler on the source code of any of the current Eiffel libraries as they use large numbers of C program calls to external routines and thus are not parallelisable.

13.3 Execution Model

There are a number of aspects that could be improved within the execution framework, all of which would improve either performance or the ease with which external code in other languages could be incorporated into a system produced using this compiler.

Use threads: The most obvious performance improvement would be to use threads, where possible, instead of *forking* processes. The *fork* facility of UNIX is, in comparison to threads, time consuming and a resource hog. Most modern operating systems include support for threads, which was not the case when this work was started.

Use fast IPC: In the same way that *fork* is wasteful, the message interchange mechanism of sockets is equally wasteful when objects need to communicate with each other on the same machine. In the same way as most modern operating systems now support threads, most also support an efficient interprocess communication mechanism (IPC). These mechanisms include primitives for shared memory and/or message exchange. Their use would greatly improve expected performance.

Incorporate an ILU interface: One of the most interesting pieces of software to have been written within this area in the past six years is ILU from Xerox Parc (Jansen et al. 1995). This enables a wrapper to be placed around programs/objects in various languages (including C, C++, Lisp, Python, etc., but not Eiffel yet) such that the ILU system can deal with inter-object communication. This may not be the most efficient solution for same-machine communications (although it performs optimisations in a way similar to that discussed in "Use fast IPC", above) but it does present a useful interface mechanism between various languages, enabling distribution. It would be a useful extension if used to support this compilation work; Eiffel parallelised code could interface with code in other programming languages, thus enabling support for "low-level" manipulations via Eiffel's external clause; manipulations that are, of necessity, missing at present. This would enable easier generation of the Eiffel programming libraries.

Avoid NFS: The execution model as it stands (see section 10.5) uses a single network file-system; this is inefficient with respect to responsiveness during execution. A simple solution to this is that instead of generating multiple binaries, i.e. one per class, the compiler could produce a single binary which pulls together all of the classes. This single binary would be placed on the network file-system and be loaded once per machine, instead of loading a new binary for each COM-based object during an object's creation. The single loading of a large object binary at program start, rather than a

number of smaller ones throughout the program, may give a slower startup time, but it should result in an improved response time during execution with its lower reliance on a relatively slow network-based filing system.

13.4 General Extensions

Inheritance: As discussed in section 9.6, there is a current area of research which is looking at the conflicts between inheritance and synchronization. Further, there are some who claim that it is really a problem with inheritance and only the symptoms can be treated (McHale 1994). This is clearly an area that needs more study before a satisfactory solution can be guaranteed within the models discussed in this thesis.

Chapter 14

Conclusion

Order and simplification are the first steps to the mastery of a subject.
The actual enemy is the unknown.
Thomas Mann

The main text in this document was broken into four parts. Part I defined the perceived problem, the thesis on which this work is based, the basic terminology and ideas within the thesis's area, and finally the objectives such that the thesis could be demonstrated. Part II, following the process model of a compiler, went through the design of a solution discussing an object-oriented model for translation which was constructed to use the theoretical model of concurrency which has been a main focus of this work. Part III discussed the structure of the implementation and was inherently - to avoid excessive length - a chapter of lists presenting inheritance hierarchies as implemented, and discussing briefly each of the classes involved, (the design of which was discussed in part II). Finally part IV is an evaluation of everything that has gone before, with a discussion of future work.

Part I introduced the problem upon which this thesis has worked:

> It should be possible to write a program in a "good" object-oriented programming language, ignoring issues resulting from a possibility of execution in a parallel environment. The derivation of this concurrency should instead be automatic, freeing a programmer to concentrate upon problem solving instead of the difficulties of implementing parallel solutions. The programs written in this language, which could utilise a parallel architecture, should be easier to write and less prone to errors, because of the decrease in complexity when compared with constructing a parallel solution. The automatic parallelisation of programs should inevitably lead to improved programmer productivity for multi-processor architectures.

This statement of the problem was further encapsulated as a thesis:

> A theoretical concurrent model of execution can be derived such that an automatic parallelisation process of a useful subset of Eiffel programs, written using certain programming principles, is possible. The parallelised program and concurrent model are unlikely to be optimal, but with sufficient refinement - post thesis study - will achieve a potential performance improvement over that of a sequential system.

Part I went on to present a survey of the area as it existed during the early part of this work. This presentation was limited to aspects that related to the above thesis and was not all-encompassing in any of the three areas discussed: object-oriented concepts, parallelism concepts, and object-oriented handling of parallelism.

Part II constructed the two central models: the theoretical model of concurrency (the COM-based model), and a model for the implementation of a compiler such that, along with the model of concurrency, the thesis could be demonstrated.

Given the thesis above, it has been demonstrated through the combination of argument and the partial implementation of a compiler for an Eiffel subset designed to utilise the COM-model approach, that there exists at least one theoretical model of concurrency along with at least one compiler implementation model which enables an object-oriented program to be transformed from its sequential form, containing no explicit parallelism, into a parallelised version automatically, such that it is capable of execution on architectures ranging from totally distributed systems to machines with only one processor.

The two models, of necessity, were more limited in scope than they could have been; this was to make the work undertaken practical and to enable completion within a limited time scale. The models have been designed such that they can be refined, as discussed in chapter 13, thus enabling a useful tool for programming parallel systems.

It may be asked *where does this thesis fit into the general picture of computer science?* It fits within the area of object-oriented programming, it does not present a panacea; this solution, even when refined as discussed in chapter 13, will not be suitable for those who need explicit control of the parallel behaviour. It does, however, demonstrate the potential for automatically parallelised programs which can be executed upon distributed systems without programmer concern on how to derive or manage those parallel aspects.

This thesis has contributed in the area of compilation in combining attribute grammar style processing within an object-oriented framework. It has demonstrated a sound theoretical model of concurrency, devoid of deadlock and starvation and yet fair in processing, which provides a practical target for the parallelisation process. The method described to achieve this absence of deadlock is a strategy involving the allocation of a unique number to each resource (or in the context of this work, each COM); each number results from an enumeration

of a COM's position in computational space, and is the basis for a linear locking of required objects by methods. As demonstrated by Havender (1968), an absence of deadlock is thus ensured.

Further to the above, the level of parallelism is improved with the application of programming principles that would be stressed on any software engineering course in Eiffel. It was also shown, as an aside to the main thesis, that there is an apparent semantic flaw within Eiffel which can be dealt with using a minimal amount of extra analysis over that required to perform a standard Eiffel compilation.

The effectiveness of the COM-based execution models presented as discussed in section 12.2.4 are unlikely even with optimisations to prove effective for general object-oriented programs. The ratio of the message-passing overhead to the actual execution of a method is in general too high. The execution strategy presented within this work is thus applicable to only a specific set of problems that achieve a better ratio. Such potential problems include the compiler written for this work, where the methods which analyse large constructs such as classes could be performed in parallel.

The final conclusion, therefore, is that the original thesis has been partly demonstrated. The automatic parallelisation has been demonstrated through the included examples. However, the potential performance improvements in the parallelised systems seem unlikely, even with optimisations, except on a very limited subset of object-oriented programs.

One never notices what has been done; one can only see what remains to be done.

Marie Curie in a letter 1894

Bibliography

Ada 95 Reference Manual (n.d.).

Agha, G. & Hewitt, C. (1987), Actors: A conceptual foundation for concurrent object-oriented programming, *in* B. Shriver & P. Wegner, eds, 'Research Directions in Object-Oriented Programming', MIT, pp. 49–74.

America, P. & Rutten, J. (1990), A layered semantics for a parallel object-oriented language, *in* J. W. de Bakker, W. P. de Roever & G. Rozenberg, eds, 'Foundations of Object-Oriented Languages', Springer Verlag, pp. 91–123.

America, P. & van der Linden, F. (1990), A parallel object-oriented language with inheritance and subtyping, *in* 'Proceedings of ECOOP/OOPSLA', ECOOP/OOPSLA, pp. 161–168.

Andrews, G. R. (1991), *Concurrent Programming Principles and Practice*, Benjamin Cummins.

Andrews, G. R. & Schneider, F. B. (1983), 'Concepts and notations for concurrent programming', *ACM Computing Surveys* **15**(1), 3–43.

Atkinson, C., Goldsack, S., Di Maio, A. & Banyan, R. (1991), 'Object-oriented concurrency and distribution in dragoon', *Journal of Object-Oriented Programming* pp. 11–18.

Avotins, J., Mingins, C. & Schmidt, H. (1995), Yoocc: Yes! an object-oriented compiler compiler, *in* 'TOOLS USA '95'. Also available as a Technical Report at http://insect.sd.monash.edu.au/.

Bahr, P. A. (1995), 'Monitor classification', *ACM Computer Surveys* **27**(1), 63–108.

Ben-Ari, B. (1982), *Principles of Concurrent Programming*, Prentice Hall.

Black, A., Hutchinson, N., Levy, H. & Carter, L. (1987), 'Distribution and abstract types in emerald', *IEEE Transactions on Software Engineering* **13**(1), 65–76.

Blair, G. S., J.Gallagher, J. & Malik, J. (1989), 'Genericity vs inheritance vs delegation vs conformance vs ...', *Journal of Object-Oriented Programming* pp. 11–17.

Blumfore, R. D., Joerg, C. F., Kuszmaul, B. C., Leirerson, C. E., Randall, K. H. & Zhou, Y. (1995), 'Clik: An efficient multithreaded runtime system', **30**(8), 207–216.

Brinch Hansen, P. (1975), 'The programming language concurrent pascal', *IEEE Transactions Software Engineering* **1**(2), 199–206.

Brinch Hansen, P. (1978), 'Distributed processes: A concurrent programming concept', *Communications of the ACM* **21**(11), 934–941.

Brinch Hansen, P. (1981*a*), 'The design of edison', *Software Prcatice and Experience* **11**(4), 363–396.

Brinch Hansen, P. (1981*b*), 'Edison: a multiprocessor language', *Software Prcatice and Experience* **11**(4), 325–361.

Briot, J.-P. & Ratuld, J. (1989), 'Design of a distributed implementation of abcl/1', *SIGPLAN NOTICES Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming* **24**(4), 15–17.

Brunskill, D. A., Rann, D. & Turner, J. H. (1995), Model quality indicators, *in* 'Third International Conference on Software Quality Management (SQM 95)', Seville, Spain.

Burkowski, F. J., Cormack, G. V. & Dueck, G. P. (1989), Architectural support for synchronous task communication, *in* ACM, ed., 'Third International Conference on Architectural Support for Programming, Boston Masachusetts Languages and Operating Systems', pp. 40–53.

Campbell, R., Russo, V. & Johnston, G. (1987), The design of a multiprocessor operating system, *in* 'USENIX C++ Conference', USENIX Association.

Caromel, D. (1989), 'Service, asynchrony and wait by necessity', *Journal of Object-Oriented Programming* pp. 12–22.

Caromel, D. (1990), 'Concurrency and reusability: From sequential to parallel', *Journal of Object-Oriented Programming* pp. 34–42.

Cheriton, D. R., Malcom, M. A., Melen, L. S. & Sager, G. R. (1979), 'Thoth, a portable real-time operating system', *Communications of the ACM* **22**(2), 105–115.

Chin, R. S. & Chanson, S. T. (1991), 'Distributed object-based programming systems', *ACM Computing Surveys* **23**(1), 71–124.

Dahl, O. J. & Nygaard, K. (1966), 'Simula, an algol-based simulation language', *Communications of the ACM* .

Dasgupta, P. (1986), A probe-based monitoring scheme for an object-oriented, distributed operating system., *in* 'Proceedings of Object-Oriented Programming Systems, Languages and Applications', ACM.

Davie, A. J. T. & Morrison, R. (1981), *Recursive Descent Compiling*, Ellis Horwood.

De Michael, L. G. & Gabriel, R. (1987), The commonlisp object system: an overview, *in* 'Proceedings of ECOOP 1987', pp. pp 201–220.

Dietel, H. (1990), *Operating Systems*, Addison Wesley.

Dijkstra, E. W. (1965), 'Solution of a problem in concurrent programming control', *Communications of the ACM* **8**(9), 569.

Dijkstra, E. W. (1968), Cooperating sequential processes, *in* F. Genuys, ed., 'Programming Languages', Academic Press, pp. 43–112.

Eliens, A. (1994), *Principles of Object-Oriented Software Development*, Addison Wesley.

Fischer, C. N. & LeBlanc, J. R. J. (1988), *Crafting a Compiler*, The Benjamin/Cummings Publishing Company, Inc.

Gentleman, W. M. (1981), 'Message passing between sequential processes: The reply primitive and administrator concept', *Software Practice and Experience* **11**, 435–466.

Gleick, J. (1987), *CHAOS Making a New Science*, Sphere books Ltd.

Goldberg, A. & Robson, D. (1989), *Smalltalk-80 The Language*, Addison and Wesley.

Grosch, J. & Emmelmann, H. (1990), A tool box for compiler construction, *in* 'Lecture Notes in Computer Science', Vol. 477, Springer Verlag.

Gunaseelan, L. & LeBlanc, Jr, R. J. (1992), Distributed eiffel: A language for programming multi-granular distributed objects on the clouds operating system, *in* 'Proceedings of the 4th International Conference on Computing Languages'.

Havender, J. W. (1968), 'Avoiding deadlock in multitasking systems', *IBM Systems Journal* .

Henderson-Sellers, B. & Edwards, J. M. (1990), 'The object-oriented systems life cycle', *Communications of the ACM* **33**(9), 143–159.

Hewitt, C. (1977), 'Viewing control structures as patterns of passing messages', *Artificial Intelligence* **8**, pp. 323–364.

Hillman, M. F. (1990), A network programming package in eiffel, *in* 'TOOLS 2', pp. 541–551.

Hoare, C. A. R. (1974), 'Monitors: An operating system concept', *Communications of the ACM* **17**(10), 549–557.

Hoare, C. A. R. (1985), *Communicating Sequential Processes*, Prentice Hall.

Hucklesy, P. & Meyer, B. (1989), The eiffel object-oriented parsing library, *in* 'TOOLS '89', pp. 501–507.

Ishikawa, Y., Tokuda, H. & Mercer, C. W. (90), Object-oriented real-time language design: Constructs for timing constraints, *in* 'ECOOP/OOPSLA '90 Proceedings', ECOOP/OOPSLA, pp. 289–298.

Jansen, B., Severson, D. & Spreitzer, M. (1995), *ILU 1.7 Reference Manual*, Xerox Corporation.

Jezequel, J. M. (1993), 'Epee: An eiffel environment to program distributed memory parallel computers', *Journal of Object-Oriented Programming* pp. 48–54.

Knuth, D. E. (1968), 'Semantics of context-free languages', *Mathematical Systems Theory* .

Lea, R. & Weightman, J. (91), Supporting object oriented languages in a distributed environment: The cool approach, *in* 'TOOLS USA '91', pp. 37–47.

Lieberman, H. (1987), Concurrent object-oriented programming in act1, *in* A. Yonezawa & M. Tokoro, eds, 'Object-Oriented Concurrent Programming', MIT press, pp. 9–36.

Liskov, B. & Scheifler, R. (1983), 'Guardians and actions: linguistic support for robust, distributed programs.', **5**(3), 381–404.

Lunau, C. P. (1989), 'Seperation of hierarchies in duo-talk', *Journal of Object-Oriented Programming* pp. 20–25.

Magnusson, B. (1990), 'A process view of objects', *Journal of Object-Oriented Programming* pp. 69–73.

Matsuoka, S., Wakita, K. & Yonezawa, A. (1993), Inheritance anomaly in object-oriented concurrent programming languages, *in* G. Agha, P. Wegner & A. Yonezawa, eds, 'Research Directions in Object-Based Concurrency', MIT Press.

May, D. (1983), 'Occam', *SIGPLAN* **18**(4), 69–79.

May, D. (1987), Occam 2 language definition, Technical report, Inmos Ltd., Bristol, United Kingdom.

McHale, C. (1994), Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance, PhD thesis, University of Dublin, Trinity College.

Meyer, B. (1988), *Object-oriented Software Construction*, Prentice Hall.

Meyer, B. (1989*a*), *Eiffel: The Language*, Interactive Software Engineering Inc.

Meyer, B. (1989*b*), The new culture of software development: Reflections on the practice of object-oriented design, *in* 'TOOLS '89', pp. 13–23.

Meyer, B. (1990), 'The new culture of software development', *Journal of Object-Oriented Programming* pp. 76–81.

Meyer, B. (1992), *Eiffel: The Language*, Prentice Hall.

Meyer, B. (1993), 'Systematic concurrent object-oriented programming', *Communications of the ACM* **36**(9), 56–80.

Meyer, B. & Nerson, J.-M. (1990*a*), *Eiffel: The Libraries*, Interactive Software Engineering Inc.

Meyer, B. & Nerson, J.-M. (1990*b*), *Eiffel: The Libraries*, 2.3 edn, Interactive Software Engineering Inc.

Micallef, J. (1988), 'Encapsulation, reusability and extensibility in object-oriented programming languages', *Journal of Object-Oriented Programming* pp. 12–34.

Nierstrasz, O. M. (1987), Active objects in hybrid, *in* 'OOPSLA87', OOPSLA, pp. 243–253.

Nygaard, K. & Dahl, O. J. (1978), 'History of programming languages conference', *SIGPLAN Notices* **13**(8), 319–340.

Rann, D., Turner, J. & Whitworth, J. (1994), *Z: A Beginner's Guide*, Chapman and Hall.

Raynal, M. (1988), *Distributed Algorithms and Protocols*, John Wiley and Sons.

Snyder, A. (1987), Inheritance and the development of encapsulated software systems, *in* B. Shriver & P. Wegner, eds, 'Research Directions in Object-Oriented Programming', MIT Press, pp. 165–188.

Steel, D. (1991), Distributed object oriented programming mechanism and experience, *in* 'TOOLS USA '91', pp. 27–35.

Tanenbaum, A. S. & Mullender, S. J. (1989/90), An introduction to amoeba, Technical report, Vrije Universiteit. This article is post-mid 89 and pre-May 90.

Theaker, C. J. & Brookes, G. R. (1993), *Concepts of Operating Systems*, Macmillan Press Ltd. pp 117-118 Deadlock Prevention.

Tilby, A. (1992), *Science and the SOUL New Cosmology, the self and God*, SPCK.

Wegner, P. (1987), The object-oriented classification paradigm, *in* B. Shriver & P. Wegner, eds, 'Research Directions in Object-Oriented Programming', MIT Press, pp. 479–560.

Wegner, P. (1990), 'Concepts and paradigms of object-oriented programming', *OOPS Messenger* **1**(1), 8–87.

Wegner, P. & Smolka, S. A. (1983), 'Processes, tasks and monitors: A comparative study of concurrent programming primitives', *IEEE Transactions on Software Engineering* pp. 446–462.

Welsh, J. & Lister, A. (1981), 'A comparative study of task communication in ada', *Software Practice and Experience* **11**, 257–290.

Wirfs-Brock, A. & Wilkerson, B. (1989), 'Variables limit reusability', *Journal of Object-Oriented Programming* pp. 34–40.

Wirth, N. (1977), 'Modula: A language for modular multi-programming', *Software Practice and Experience* **7**, 3–35.

Wirth, N. (1982), *Programming in Modula-2*, Springer-Verlag.

Wolff, T. (1995), Transparently distributing objects with inheritance, *in* '28th Hawaii International Conference on System Sciences', pp. 279–303.

Yokote, Y. & Tokoro, M. (1987), Concurrent programming in concurrentsmalltalk, *in* 'Object-Oriented Concurrent Programming', MIT Press.

Yonezawa, A., Briot, J.-P. & Shibayama, E. (1986), Object-oriented concurrent programming in abcl/1, *in* 'OOPSLA '86', pp. 258–268.

Zimmermann, H., Banino, J. S., Caristan, A., Guillemont, M. & Morisset, G. (1981), Basic concepts for the support of distributed systems: The chorus approach, *in* 'Proceedings of the 2nd International Conference on Distributed Computing Systems', IEEE, pp. 60–66.

Appendix A

Object Details

A.1 Feature level routines

The main feature level routines and attributes inside a feature object can be summarised as follows:

uses_externals: if set, this indicates that the feature makes reference to external 'C' program routines.[1]

exportable_feature: if true, this indicates that this feature would be a permissible export from a class to be parallelised, i.e. it fulfils all of the necessary restrictions of being either a COM or of simple type such as integer, real, character, etc.

is_create: if true, this indicates that this feature is the creation method feature for objects described as being of this class's type.

the_type: the type returned by a query-feature.

like_current: if true, the type returned by this query-feature will be the type of the object of which it is a member at run-time, allowing for any dynamic binding.

is_infix_method: if true, this feature is usable as an infix operator.

is_attribute: if true, this feature is a description of an attribute.

is_constant: if true, this feature is an attribute with a constant value.

is_method: if true, this feature is a definition of a method.

[1] For the purpose of this work C externals are not supported, and therefore although Eiffel has support for the idea this amounts to an error

is_query: if true, this feature is a query-method.

is_state_changer: if true, this feature causes state changes to entities external to its own definition, this may be as a result of changes to actual parameters, or because of changes to class variables.

state_formal_parameters: this indicates for each formal parameter whether the code contained within the feature, causes a change in their state.

external_accessed_vars: this is a list of variables that this feature accesses, for which it has no local definition, assumed to be class variables. There is within the *external_accessed_vars* object a mapping from the name of the variable to a boolean indicating whether or not this feature changes its state.

all_types: this is a list of types which are used within this feature, these types include the type of formal parameters, local variables, and any query-feature's type. This information enables the class to request the compilation of any classes for which it is a client.

A.2 Internal Feature Routines

The main internal feature level methods and attributes are summarised as follows:

formal_parameters: this lists the formal parameters in the order they appear in the feature's header; the name only is recorded, as the available information about any particular symbol is recorded in the *feature_symbol_table*.

accessed_vars: this feature lists all variables that are accessed within the feature, including whether or not their state is changed by any actions within the feature. The external_accessed_vars are an extracted subset of this feature.

feature_symbol_table: this lists all *symbol_names* and their corresponding *symbol_object* which records all information known about a particular symbol.

has: this checks whether there is a local definition for a symbol - either as a local variable or formal parameter.

item: this returns the *symbol_object* associated with a symbol name.

is_local_var: if true, the name supplied as a parameter is defined to be a local variable within this feature.

is_formal_parameter: if true, the name is used as a formal parameter in this feature's header.

expression_info: As the name suggests it holds information about expressions within this feature, it is built during parsing and semantic analysis and after some processing is passed back down the hierarchy to enable code generation. Specifically, during code generation the expression information is used to break up statements and to schedule them to support parallelism (see chapter 9).

A.3 Symbol Object Routines

is_written will be set if the symbol is the subject of an assignment, or a state changing method is applied to it.

is_external: true, if the symbol is defined in an external clause; i.e. it is a 'C' program routine.

is_associated: true, if the symbol is declared to have a type "*like X*" where "*X*" is another object.

associate: the name of the associated object; i.e. the object which this symbol has been described to be of the same type as.

is_assigned: true, if and only if it is the subject of an assignment.

is_local: true, if the symbol is defined to be a local variable to the current feature.

is_formal_parameter: true, if the symbol is a formal parameter to the current feature.

is_actual_parameter: true, if the symbol is used as an actual parameter to a method call.

symbol_type: will hold the type of the object when it has been derived.

is_type: true, if the symbol represents a type name.

like_current: true, if the symbol is declared as having the same type (*like current*) as the object of which it is a member.

exportable_type: true, if the symbol's type satisfies criteria for exportability (see section 6.5).

merge: this is a method that enables the merging of information about two symbols which are in actuality the same entity, referred to in different places in the program.

Appendix B

Example 1: Construct Compilation

This appendix contains example code generated by the compiler for this thesis and hence is the parallelised-code version of the class used to demonstrate the compilation of constructs 11.2.

B.1 CONSTRUCT

```
class CONSTRUCT export

        repeat COM, meth1,meth2,meth3,n,set_to,cons
inherit
        COM
                rename process_command as c_process_command
,
                Create as c_create
                redefine process_command, pre_loop, method_connect;


feature

        create is
                do
                        c_create("CONSTRUCT");
                end; -- create

        pre_loop is
                local
                        cstr : CODED_STRING
                do
                        cstr.Create;
                        r_object_gen.new_method("meth3_METHOD_COURIER",host_name,
                        0,"meth3", "meth3_METHOD");
                        r_object_gen.new_method("loop_statement_METHOD_COURIER",host_name,
                        0,"loop_statement", "loop_statement_METHOD");
                        r_object_gen.new_method("method_application_METHOD_COURIER",host_name,
                        0,"method_application", "method_application_METHOD");
```

```
                r_object_gen.new_method("if_statement_METHOD_COURIER",host_name,
                0,"if_statement", "if_statement_METHOD");
                r_object_gen.new_method("meth2_METHOD_COURIER",host_name,
                0,"meth2", "meth2_METHOD");
                r_object_gen.new_method("mk_construct_METHOD_COURIER",host_name,
                0,"mk_construct", "mk_construct_METHOD");
                r_object_gen.new_method("set_to_METHOD_COURIER",host_name,
                0,"set_to", "set_to_METHOD");
                r_object_gen.new_method("meth1_METHOD_COURIER",host_name,
                0,"meth1", "meth1_METHOD");
        end; -- pre_loop

method_connect(cmd:CODED_STRING;conn:COM_CONNECTION) is
        local
                str:STRING;
                cs : CODED_STRING;
                ms : MESSAGE;
        do
                str:=cmd.the_name;
                if str.equal("meth3_METHOD") then
                        meth3 := conn;
                elsif str.equal("loop_statement_METHOD") then
                        loop_statement := conn;
                elsif str.equal("method_application_METHOD") then
                        method_application := conn;
                elsif str.equal("if_statement_METHOD") then
                        if_statement := conn;
                elsif str.equal("meth2_METHOD") then
                        meth2 := conn;
                elsif str.equal("mk_construct_METHOD") then
                        mk_construct := conn;
                        ms.Create;
                        cs.Create;
                        cs.set_name(my_name);  -- courier required!
                        cs.com_reply(conn);
                        ms.set_method_name("mk_construct");
                        cs.set_message(ms);
                        mk_construct.send_command(cs);  -- as opposed to blocking send_msg
                elsif str.equal("set_to_METHOD") then
                        set_to := conn;
                elsif str.equal("meth1_METHOD") then
                        meth1 := conn;
                else
                        info_display.display_msg(my_name,conn.talker_name,cmd,"UNKNOWN MESSAGE'
                end; -- if
        end; -- method_connect

mk_construct : COM_CONNECTION;   --- METHOD;    -- Create


n: INTEGER;

cons: COM_CONNECTION;

if_statement : COM_CONNECTION;   --- METHOD;    -- if_statement


loop_statement : COM_CONNECTION;   --- METHOD;  -- loop_statement


method_application : COM_CONNECTION;   --- METHOD;      -- method_application


meth1 : COM_CONNECTION;   --- METHOD;   -- meth1


meth2 : COM_CONNECTION;   --- METHOD;   -- meth2
```

```
        meth3 : COM_CONNECTION;    --- METHOD;    -- meth3


        set_to : COM_CONNECTION;    --- METHOD;

process_command(cmd : like last_command, conn : COM_CONNECTION) is
        local
                local_msg : MESSAGE;
                cstr : CODED_STRING;
                l_meth3 : CONSTRUCT;
                                                                        l_meth2 : CONSTRUCT;
                                                l_meth1 : CONSTRUCT;

        do
                cstr.Create;
                if cmd.is_a_reply       then
                elsif cmd.is_a_message then
                        local_msg ?= cmd.the_message;
                        if local_msg.void then
                                io.putstring("ERR:HELLPPPP!")
                        end; -- if
                        if not check_access(conn.talker_name) then
                        -- check if this caller has access rights
                                        queue_request(cmd,conn);
                        elsif local_msg.method_name.equal("meth3") then
                                cmd.com_send(meth3);     -- query
                        elsif local_msg.method_name.equal("loop_statement") then
                                cmd.com_send(loop_statement);    -- command
                        elsif local_msg.method_name.equal("cons") then
                                cstr.set_com(cons.com_port);
                                cstr.com_reply(conn);
                        elsif local_msg.method_name.equal("method_application") then
                                cmd.com_send(method_application);        -- command
                        elsif local_msg.method_name.equal("if_statement") then
                                cmd.com_send(if_statement);      -- command
                        elsif local_msg.method_name.equal("meth2") then
                                cmd.com_send(meth2);     -- query
                        elsif local_msg.method_name.equal("n") then
                                cstr.set_integer(n);
                                cstr.com_reply(conn);
                        elsif local_msg.method_name.equal("set_to") then
                                cmd.com_send(set_to);    -- command
                        elsif local_msg.method_name.equal("meth1") then
                                cmd.com_send(meth1);     -- query
                        else
                                io.putstring("\nUNDEFINED MESSAGE IN AN INSTANCE OF");
                                io.putstring(class_name);
                                io.putstring("\n MESSAGE IS:");
                                io.putstring(cmd.string_rep);
                        end; -- if
                else
                        c_process_command(cmd,conn);
                end; -- if
         end; -- process_command

end;
```

B.2 MK_CONSTRUCT

```
class MK_CONSTRUCT export
repeat COMMAND
inherit COMMAND
rename Create as c_create
```

```
        redefine do_method
feature

        Create is
                        do
                                creation_method := True;
                                c_create ("MK_CONSTRUCT");
                        end; -- Create
        do_method  is

                local
                        i: INTEGER;
                do
                        local_comp.clear;
                        local_msg.clear;
                        local_msg.set_method_name("if_statement");
                        local_comp.extend(local_msg.coded_string);
                        master_connection.send_command(local_comp.coded_string);
                        local_comp.clear;
                        local_msg.clear;
                        local_msg.set_method_name("loop_statement");
                        local_comp.extend(local_msg.coded_string);
                        master_connection.send_command(local_comp.coded_string);
                        local_comp.clear;
                        local_msg.clear;
                        local_msg.set_method_name("method_application");
                        local_comp.extend(local_msg.coded_string);
                        master_connection.send_command(local_comp.coded_string);
                end

end; -- class MK_CONSTRUCT
```

B.3 IF_STATEMENT

```
class IF_STATEMENT export
repeat COMMAND
inherit COMMAND
rename Create as c_create
        redefine do_method, pre_method, post_method
feature

        Create is
                        do
                                c_create ("IF_STATEMENT");
                        end; -- Create

        pre_method is
                local
                        p:OBJECT_PORT;
                        cs : CODED_STRING;
                        co : OBJECT_PORT;
                do
                        cs.Create;
                        cs.set_reply("Early Return");
                        master_connection.reply_msg(cs);
                end; -- pre_method


        post_method is
                local
                        hhhh:CODED_STRING;
                do
                end; -- post_method
```

```
        do_method  is

                local
                        i: INTEGER;
                do
                        i := 0;
                        local_comp.clear;
                        local_msg.clear;
                        local_msg.set_method_name("meth1");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("n");
                        local_comp.extend(local_msg.coded_string);
                        if i - master_connection.send_msg(local_comp.coded_string).the_integer < 1 ther
                                i := i + 1;
                        else
                                local_comp.clear;
                                local_msg.clear;
                                local_msg.set_method_name("meth2");
                                local_comp.extend(local_msg.coded_string);
                                local_msg.set_method_name("n");
                                local_comp.extend(local_msg.coded_string);
                                if i + master_connection.send_msg(local_comp.coded_string).the_integer
                                        i := i * 2;
                                else
                                        local_comp.clear;
                                        local_msg.clear;
                                        local_msg.set_method_name("meth3");
                                        local_comp.extend(local_msg.coded_string);
                                        local_msg.set_method_name("n");
                                        local_comp.extend(local_msg.coded_string);
                                        if i * master_connection.send_msg(local_comp.coded_string).the_
                                                i := i - 10;
                                        else
                                                i := 1;
                                        end; -- if
                                end; -- if
                        end;
                end

end; -- class IF_STATEMENT
```

B.4 LOOP_STATEMENT

```
class LOOP_STATEMENT export
repeat COMMAND
inherit COMMAND
rename Create as c_create
        redefine do_method, pre_method, post_method
feature

        Create is
                        do
                                c_create ("LOOP_STATEMENT");
                        end; -- Create

        pre_method is
                local
                        p:OBJECT_PORT;
                        cs : CODED_STRING;
                        co : OBJECT_PORT;
                do
                        cs.Create;
                        cs.set_reply("Early Return");
                        master_connection.reply_msg(cs);
```

```
        end; -- pre_method


post_method is
        local
                hhhh:CODED_STRING;
        do
        end; -- post_method

do_method  is

        local
                i: INTEGER;
        do
                from
                        i := 1;
                        -- CONDITIONAL's extra_ass
                        local_comp.clear;
                        local_msg.clear;
                        local_msg.set_method_name("meth3");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("n");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("meth2");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("meth3");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("n");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("meth1");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("meth2");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("meth3");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("n");
                        local_comp.extend(local_msg.coded_string);

                until
                        i = master_connection.send_msg(local_comp.coded_string).the_integer
                loop
                        i := i + 1;;
                        -- CONDITIONAL's extra_ass
                        local_comp.clear;
                        local_msg.clear;
                        local_msg.set_method_name("meth3");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("n");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("meth2");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("meth3");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("n");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("meth1");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("meth2");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("meth3");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("n");
                        local_comp.extend(local_msg.coded_string);

                 end;    -- loop

          end
```

```
end; -- class LOOP_STATEMENT
```

B.5 METHODn

It should be noted that *meth1*, *meth2* and *meth3* are all the same, therefore only one of the parallelised versions is incorporated here as the others will be the same with minor naming differences.

```
class METH1 export
repeat COMMAND
inherit COMMAND
rename Create as c_create
        redefine do_method, pre_method, post_method
feature

        cons : COM_CONNECTION; -- Accessed Var of type: CONSTRUCT

        Create is
                        do
                                c_create ("METH1");
                        end; -- Create

        pre_method is
                local
                        p:OBJECT_PORT;
                        cs : CODED_STRING;
                        co : OBJECT_PORT;
                do
                        cs.Create;
                        cs.set_give_me("cons");
                        co := master_connection.send_msg(cs).the_com; -- CONSTRUCT
                        if co.port = 0 then
                                -- not created yet
                        elsif co.port = cons.com_port.port and co.host.equal(cons.com_port.host) then
                        -- no need to connect as we already have this object
                        else
                                connect_to_port(co.the_com.host,co.the_com.port);
                                cons := new_connection;
                        end; -- if
                        -- no early return as it is a query
                end; -- pre_method


        post_method is
                local
                        hhhh:CODED_STRING;
                do
                        local_code.set_object_res("cons",cons.remote_port); -- CONSTRUCT
                        local_code.com_send(master_connection);
                end; -- post_method

        do_method  is
                local
                        Reslut : COM_CONNECTION;

                -- Generated locals
                        l1 : COM_CONNECTION;
                        cstr : CODED_STRING;

                do
                        local_comp.clear;
```

```
                        local_msg.clear;
                        l1 := cons; -- message value generated (break on a non-COM)
                        Reslut := l1;
                        cstr.Create;
                        cstr.set_com(Reslut.com_port);
                        cstr.com_reply(master_connection);

                end

end; -- class METH1
```

B.6 METHOD_APPLICATION

```
class METHOD_APPLICATION export
repeat COMMAND
inherit COMMAND
rename Create as c_create
        redefine do_method, pre_method, post_method
feature

        cons : COM_CONNECTION; -- Accessed Var of type: CONSTRUCT

        Create is
                        do
                                c_create ("METHOD_APPLICATION");
                        end; -- Create

        pre_method is
                local
                        p:OBJECT_PORT;
                        cs : CODED_STRING;
                        co : OBJECT_PORT;
                do
                        cs.Create;
                        cs.set_give_me("cons");
                        co := master_connection.send_msg(cs).the_com; -- CONSTRUCT
                        if co.port = 0 then
                                -- not created yet
                        elsif co.port = cons.com_port.port and co.host.equal(cons.com_port.host) then
                        -- no need to connect as we already have this object
                        else
                                connect_to_port(co.the_com.host,co.the_com.port);
                                cons := new_connection;
                        end; -- if
                        cs.set_reply("Early Return");
                        master_connection.reply_msg(cs);
                end; -- pre_method


        post_method is
                local
                        hhhh:CODED_STRING;
                do
                        local_code.set_object_res("cons",cons.remote_port); -- CONSTRUCT
                        local_code.com_send(master_connection);
                end; -- post_method

        do_method  is
                local
                do
                        local_comp.clear;
                        local_msg.clear;
                        local_msg.set_method_name("meth3");
                        local_comp.extend(local_msg.coded_string);
```

```
                        local_msg.set_method_name("set_to");
                        local_code.set_integer(4);
                        local_msg.extend(local_code);
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("meth2");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("meth3");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("set_to");
                        local_code.set_integer(4);
                        local_msg.extend(local_code);
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("meth1");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("meth2");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("meth3");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("set_to");
                        local_code.set_integer(4);
                        local_msg.extend(local_code);
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("meth1");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("meth2");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("meth3");
                        local_comp.extend(local_msg.coded_string);
                        local_msg.set_method_name("set_to");
                        local_code.set_integer(4);
                        local_msg.extend(local_code);
                        local_comp.extend(local_msg.coded_string);
                        cons.send_command(local_comp.coded_string);
                end

end; -- class METHOD_APPLICATION
```

Appendix C

Example 2: Producer-Consumer

This appendix contains example code generated by the compiler produced for this thesis and hence is the parallelised-code versions of the classes describing the producer-consumer problem in section 11.4.

C.1 BOUND class

C.1.1 BOUND

```
class BOUND
inherit
   COM
      rename process_command as c_process_command
,
      Create as c_create
      redefine process_command, pre_loop, method_connect;


feature

      b: COM_CONNECTION;

      producer: COM_CONNECTION;

      consumer: COM_CONNECTION;

   create is
      do
         c_create("BOUND");
      end; -- create

   pre_loop is
      local
         cstr : CODED_STRING
      do
         cstr.Create;
         r_object_gen.new_method("mk_bound_METHOD_COURIER",host_name,
         0,"mk_bound", "mk_bound_METHOD");
      end; -- pre_loop
```

```
   method_connect(cmd:CODED_STRING;conn:COM_CONNECTION) is
      local
         str:STRING;
         cs : CODED_STRING;
         ms : MESSAGE;
      do
         str:=cmd.the_name;
         if str.equal("mk_bound_METHOD") then
            mk_bound := conn;
            ms.Create;
            cs.Create;
            cs.set_name(my_name);  -- courier required!
            cs.com_reply(conn);
            ms.set_method_name("mk_bound");
            cs.set_message(ms);
            mk_bound.send_command(cs);  -- as opposed to blocking send_msg
         else
            info_display.display_msg(my_name,conn.talker_name,cmd,"UNKNOWN MESSAGE");
         end; -- if
      end; -- method_connect

   mk_bound : COM_CONNECTION;   --- METHOD;

   process_command(cmd : like last_command, conn : COM_CONNECTION) is
      do
         c_process_command(cmd, conn);
      end; -- process_command

end;  -- BOUND
```

C.1.2 MK_BOUND

```
class MK_BOUND export
repeat COMMAND
inherit COMMAND
rename Create as c_create
   redefine do_method
feature

   consumer : COM_CONNECTION; -- Accessed Var of type: CONSUMER

   producer : COM_CONNECTION; -- Accessed Var of type: PRODUCER

   b : COM_CONNECTION; -- Accessed Var of type: BUFFER

   Create is
         do
            creation_method := True;
            c_create ("MK_BOUND");
         end; -- Create
   do_method  is
      local
      do
         new_obj.set_object("producer_COURIER",host_name, 0,"producer","producer");
         local_code.set_new_object(new_obj);
         local_code.com_send(master_connection);
         connect_to_port(local_code.the_com.host, local_code.the_com.port);
         producer := new_connection;
         producer.set_object_port(local_code.the_com);
;
         new_obj.set_object("consumer_COURIER",host_name, 0,"consumer","consumer");
         local_code.set_new_object(new_obj);
         local_code.com_send(master_connection);
         connect_to_port(local_code.the_com.host, local_code.the_com.port);
         consumer := new_connection;
```

```
        consumer.set_object_port(local_code.the_com);
;
        from
           local_comp.clear;
           local_msg.clear;
           local_msg.set_method_name("produce");
           local_comp.extend(local_msg.coded_string);
           producer.send_command(local_comp.coded_string);
           local_comp.clear;
           local_msg.clear;
           local_comp.clear;
           local_msg.clear;
           local_msg.set_method_name("lastprod");
           local_comp.extend(local_msg.coded_string);
           local_msg.set_method_name("append");
           local_code.set_integer(producer.send_msg(local_comp.coded_string).the_integer);
           local_msg.extend(local_code);
           local_comp.extend(local_msg.coded_string);
           b.send_command(local_comp.coded_string);
        until
           False
        loop
           if not b.send_msg(local_comp.coded_string).the_boolean then
              local_comp.clear;
              local_msg.clear;
              local_comp.clear;
              local_msg.clear;
              local_msg.set_method_name("item");
              local_comp.extend(local_msg.coded_string);
              local_msg.set_method_name("consume");
              local_code.set_integer(b.send_msg(local_comp.coded_string).the_integer);
              local_msg.extend(local_code);
              local_comp.extend(local_msg.coded_string);
              consumer.send_command(local_comp.coded_string);
              local_comp.clear;
              local_msg.clear;
              local_msg.set_method_name("take");
              local_comp.extend(local_msg.coded_string);
              b.send_command(local_comp.coded_string);
           end;  -- if

           if not b.send_msg(local_comp.coded_string).the_boolean then
              local_comp.clear;
              local_msg.clear;
              local_msg.set_method_name("produce");
              local_comp.extend(local_msg.coded_string);
              producer.send_command(local_comp.coded_string);
              local_comp.clear;
              local_msg.clear;
              local_comp.clear;
              local_msg.clear;
              local_msg.set_method_name("lastprod");
              local_comp.extend(local_msg.coded_string);
              local_msg.set_method_name("append");
              local_code.set_integer(producer.send_msg(local_comp.coded_string).the_integer);
              local_msg.extend(local_code);
              local_comp.extend(local_msg.coded_string);
              b.send_command(local_comp.coded_string);
           end;;
        end;
     end

end; -- class MK_BOUND
```

C.2 PRODUCER class

C.2.1 Producer

```
class PRODUCER export

   repeat COM, produce,lastprod
inherit
   COM
      rename process_command as c_process_command
,
      Create as c_create
      redefine process_command, pre_loop, method_connect;


feature

   create is
      do
         c_create("PRODUCER");
      end; -- create

   pre_loop is
      local
         cstr : CODED_STRING
      do
         cstr.Create;
         r_object_gen.new_method("produce_METHOD_COURIER",host_name,
         0,"produce", "produce_METHOD");
      end; -- pre_loop

   method_connect(cmd:CODED_STRING;conn:COM_CONNECTION) is
      local
         str:STRING;
         cs : CODED_STRING;
         ms : MESSAGE;
      do
         str:=cmd.the_name;
         if str.equal("produce_METHOD") then
            produce := conn;
         else
            info_display.display_msg(my_name,conn.talker_name,cmd,"UNKNOWN MESSAGE");
         end; -- if
      end; -- method_connect

   produce : COM_CONNECTION;    --- METHOD;    -- produce


   lastprod: INTEGER;

process_command(cmd : like last_command, conn : COM_CONNECTION) is
   local
      local_msg : MESSAGE;
      cstr : CODED_STRING;

   do
      cstr.Create;
      if cmd.is_a_reply then
      elsif cmd.is_a_message then
         local_msg ?= cmd.the_message;
         if local_msg.void then
            io.putstring("ERR:HELLPPPP!")
         end; -- if
         if not check_access(conn.talker_name) then
         -- check if this caller has access rights
```

```
            queue_request(cmd,conn);
         elsif local_msg.method_name.equal("produce") then
            cmd.com_send(produce);  -- command
         elsif local_msg.method_name.equal("lastprod") then
            cstr.set_integer(lastprod);
            cstr.com_reply(conn);
         else
            io.putstring("\nUNDEFINED MESSAGE IN AN INSTANCE OF");
            io.putstring(class_name);
            io.putstring("\n MESSAGE IS:");
            io.putstring(cmd.string_rep);
         end; -- if
      else
         c_process_command(cmd,conn);
      end; -- if
   end; -- process_command

end;  -- class PRODUCER
```

C.2.2 Produce

```
class PRODUCE export
repeat COMMAND
inherit COMMAND
rename Create as c_create
   redefine do_method, pre_method, post_method
feature

   Create is
         do
            c_create ("PRODUCE");
         end; -- Create

   pre_method is
      local
         p:OBJECT_PORT;
         cs : CODED_STRING;
         co : OBJECT_PORT;
      do
         cs.Create;
         cs.set_reply("Early Return");
         master_connection.reply_msg(cs);
      end; -- pre_method


   post_method is
      local
         hhhh:CODED_STRING;
      do
      end; -- post_method

   do_method  is
         -- does whatever is required to produce
local
      do -- generate data and put it into
         -- lastprod

      end

end; -- class PRODUCE
```

C.3 CONSUMER class

C.3.1 Consumer

```
class CONSUMER export

   repeat COM, consume
inherit
   COM
      rename process_command as c_process_command
,
      Create as c_create
      redefine process_command, pre_loop, method_connect;


feature

   create is
      do
         c_create("CONSUMER");
      end; -- create

   pre_loop is
      local
         cstr : CODED_STRING
      do
         cstr.Create;
         r_object_gen.new_method("consume_METHOD_COURIER",host_name,
         0,"consume", "consume_METHOD");
      end; -- pre_loop

   method_connect(cmd:CODED_STRING;conn:COM_CONNECTION) is
      local
         str:STRING;
         cs : CODED_STRING;
         ms : MESSAGE;
      do
         str:=cmd.the_name;
         if str.equal("consume_METHOD") then
            consume := conn;
         else
            info_display.display_msg(my_name,conn.talker_name,cmd,"UNKNOWN MESSAGE");
         end; -- if
      end; -- method_connect

   consume : COM_CONNECTION;   --- METHOD -- consume
;

process_command(cmd : like last_command, conn : COM_CONNECTION) is
   local
      local_msg : MESSAGE;
      cstr : CODED_STRING;

   do
      cstr.Create;
      if cmd.is_a_reply then
      elsif cmd.is_a_message then
         local_msg ?= cmd.the_message;
         if local_msg.void then
            io.putstring("ERR:HELLPPPP!")
         end; -- if
         if not check_access(conn.talker_name) then
         -- check if this caller has access rights
               queue_request(cmd,conn);
         elsif local_msg.method_name.equal("consume") then
```

```
            cmd.com_send(consume);   -- command
         else
            io.putstring("\nUNDEFINED MESSAGE IN AN INSTANCE OF");
            io.putstring(class_name);
            io.putstring("\n MESSAGE IS:");
            io.putstring(cmd.string_rep);
         end; -- if
      else
         c_process_command(cmd,conn);
      end; -- if
   end; -- process_command

end;
```

C.3.2 Consume

```
class CONSUME export
repeat COMMAND
inherit COMMAND
rename Create as c_create
   redefine do_method, pre_method, post_method
feature

   v : INTEGER;

   Create is
         do
            c_create ("CONSUME");
         end; -- Create

   pre_method is
      local
         p:OBJECT_PORT;
         cs : CODED_STRING;
         co : OBJECT_PORT;
      do
         cs.Create;
         v := msg.i_th(1).the_integer;
         cs.set_reply("Early Return");
         master_connection.reply_msg(cs);
      end; -- pre_method


   post_method is
      local
         hhhh:CODED_STRING;
      do
      end; -- post_method

   do_method  is
      local
      do -- code to make use of value v

      end

end; -- class CONSUME
```

C.4 BUFFER class

C.4.1 MK_BUFFER

```
class MK_BUFFER export
repeat COMMAND
inherit COMMAND
rename Create as c_create
   redefine do_method
feature

   Create is
         do
            creation_method := True;
            c_create ("MK_BUFFER");
         end; -- Create
   do_method  is
      local
      do -- Allocate(0,BUFFERSIZE); -- sets up buffer

      end

end; -- class MK_BUFFER
```

C.4.2 ISEMPTY

```
class ISEMPTY export
repeat COMMAND
inherit COMMAND
rename Create as c_create
   redefine do_method, pre_method, post_method
feature

   n : INTEGER; -- Accessed Var of type: INTEGER

   Create is
         do
            c_create ("ISEMPTY");
         end; -- Create

   pre_method is
      local
         p:OBJECT_PORT;
         cs : CODED_STRING;
         co : OBJECT_PORT;
      do
         cs.Create;
         cs.set_give_me("n");
         n := master_connection.send_msg(cs).the_integer;
         -- no early return as it is a query
      end; -- pre_method


   post_method is
      local
         hhhh:CODED_STRING;
      do
         hhhh.Create;
         hhhh.set_integer(n);
         local_code.set_object_res("n",hhhh);
         local_code.com_send(master_connection);
      end; -- post_method
```

```
   do_method  is
      local
         Reslut : BOOLEAN;

      -- Generated locals
         cstr : CODED_STRING;

      do
         Reslut := n = 0;
         cstr.Create;
         cstr.set_boolean(Reslut);
         cstr.com_reply(master_connection);

      end

end; -- class ISEMPTY
```

C.4.3 ISFULL

```
class ISFULL export
repeat COMMAND
inherit COMMAND
rename Create as c_create
   redefine do_method, pre_method, post_method
feature

   n : INTEGER; -- Accessed Var of type: INTEGER

   BUFFERSIZE : INTEGER; -- Accessed Var of type: INTEGER

   Create is
         do
            c_create ("ISFULL");
         end; -- Create

   pre_method is
      local
         p:OBJECT_PORT;
         cs : CODED_STRING;
         co : OBJECT_PORT;
      do
         cs.Create;
         cs.set_give_me("n");
         n := master_connection.send_msg(cs).the_integer;
         cs.set_give_me("BUFFERSIZE");
         BUFFERSIZE := master_connection.send_msg(cs).the_integer;
         -- no early return as it is a query
      end; -- pre_method


   post_method is
      local
         hhhh:CODED_STRING;
      do
         hhhh.Create;
         hhhh.set_integer(n);
         local_code.set_object_res("n",hhhh);
         local_code.com_send(master_connection);
         hhhh.Create;
         hhhh.set_integer(BUFFERSIZE);
         local_code.set_object_res("BUFFERSIZE",hhhh);
         local_code.com_send(master_connection);
      end; -- post_method
```

```
   do_method  is
      local
         Reslut : BOOLEAN;

      -- Generated locals
         cstr : CODED_STRING;

      do
         Reslut := n = BUFFERSIZE;
         cstr.Create;
         cstr.set_boolean(Reslut);
         cstr.com_reply(master_connection);

      end

end; -- class ISFULL
```

C.4.4 PUT

```
class PUT export
repeat COMMAND
inherit COMMAND
rename Create as c_create
   redefine do_method, pre_method, post_method
feature

   in : INTEGER;

   v : INTEGER;

   Create is
         do
            c_create ("PUT");
         end; -- Create

   pre_method is
      local
         p:OBJECT_PORT;
         cs : CODED_STRING;
         co : OBJECT_PORT;
      do
         cs.Create;
         in := msg.i_th(1).the_integer;
         v := msg.i_th(2).the_integer;
         cs.set_reply("Early Return");
         master_connection.reply_msg(cs);
      end; -- pre_method


   post_method is
      local
         hhhh:CODED_STRING;
      do
      end; -- post_method

   do_method  is
      local
      do -- code to actually put value in the buffer

      end

end; -- class PUT
```

C.4.5 APPEND

```
class APPEND export
repeat COMMAND
inherit COMMAND
rename Create as c_create
   redefine do_method, pre_method, post_method
feature

   n : INTEGER; -- Accessed Var of type: INTEGER

   BUFFERSIZE : INTEGER; -- Accessed Var of type: INTEGER

   in : INTEGER; -- Accessed Var of type: INTEGER

   v : INTEGER;

   Create is
         do
            c_create ("APPEND");
         end; -- Create

   pre_method is
      local
         p:OBJECT_PORT;
         cs : CODED_STRING;
         co : OBJECT_PORT;
      do
         cs.Create;
         v := msg.i_th(1).the_integer;
         cs.set_give_me("n");
         n := master_connection.send_msg(cs).the_integer;
         cs.set_give_me("BUFFERSIZE");
         BUFFERSIZE := master_connection.send_msg(cs).the_integer;
         cs.set_give_me("in");
         in := master_connection.send_msg(cs).the_integer;
         cs.set_reply("Early Return");
         master_connection.reply_msg(cs);
      end; -- pre_method


   post_method is
      local
         hhhh:CODED_STRING;
      do
         hhhh.Create;
         hhhh.set_integer(n);
         local_code.set_object_res("n",hhhh);
         local_code.com_send(master_connection);
         hhhh.Create;
         hhhh.set_integer(BUFFERSIZE);
         local_code.set_object_res("BUFFERSIZE",hhhh);
         local_code.com_send(master_connection);
         hhhh.Create;
         hhhh.set_integer(in);
         local_code.set_object_res("in",hhhh);
         local_code.com_send(master_connection);
      end; -- post_method

   do_method  is
      local
      do
         local_comp.clear;
         local_msg.clear;
         local_msg.set_method_name("put");
         local_code.set_integer(in);
```

```
         local_msg.extend(local_code);
         local_code.set_integer(v);
         local_msg.extend(local_code);
         local_comp.extend(local_msg.coded_string);
         master_connection.send_command(local_comp.coded_string);
         in := in + 1;
         if in = BUFFERSIZE then
            in := 1; -- Eiffel arrays start at 1

         end;  -- if

         n := n + 1;
      end

end; -- class APPEND
```

C.4.6 GET

```
class GET export
repeat COMMAND
inherit COMMAND
rename Create as c_create
   redefine do_method, pre_method, post_method
feature

   place : INTEGER;

   Create is
         do
            c_create ("GET");
         end; -- Create

   pre_method is
      local
         p:OBJECT_PORT;
         cs : CODED_STRING;
         co : OBJECT_PORT;
      do
         cs.Create;
         place := msg.i_th(1).the_integer;
         -- no early return as it is a query
      end; -- pre_method


   post_method is
      local
         hhhh:CODED_STRING;
      do
      end; -- post_method

   do_method  is
      local
         Reslut : INTEGER;

      -- Generated locals
         cstr : CODED_STRING;

      do -- selects the item at point in the array

         cstr.Create;
         cstr.set_integer(Reslut);
         cstr.com_reply(master_connection);

      end
```

```
end; -- class GET
```

C.4.7 ITEM

```
class ITEM export
repeat COMMAND
inherit COMMAND
rename Create as c_create
   redefine do_method, pre_method, post_method
feature

   next_out : INTEGER; -- Accessed Var of type: INTEGER

   Create is
         do
            c_create ("ITEM");
         end; -- Create

   pre_method is
      local
         p:OBJECT_PORT;
         cs : CODED_STRING;
         co : OBJECT_PORT;
      do
         cs.Create;
         cs.set_give_me("next_out");
         next_out := master_connection.send_msg(cs).the_integer;
         -- no early return as it is a query
      end; -- pre_method


   post_method is
      local
         hhhh:CODED_STRING;
      do
         hhhh.Create;
         hhhh.set_integer(next_out);
         local_code.set_object_res("next_out",hhhh);
         local_code.com_send(master_connection);
      end; -- post_method

   do_method  is
         -- returns the value the next_out item
local
         Reslut : INTEGER;

      -- Generated locals
         cstr : CODED_STRING;

      do
         local_comp.clear;
         local_msg.clear;
         local_msg.set_method_name("get");
         local_code.set_integer(next_out);
         local_msg.extend(local_code);
         local_comp.extend(local_msg.coded_string);
         Reslut := master_connection.send_msg(local_comp.coded_string).the_integer;
         cstr.Create;
         cstr.set_integer(Reslut);
         cstr.com_reply(master_connection);

      end

end; -- class ITEM
```

C.4.8 TAKE

```
class TAKE export
repeat COMMAND
inherit COMMAND
rename Create as c_create
   redefine do_method, pre_method, post_method
feature

   n : INTEGER; -- Accessed Var of type: INTEGER

   BUFFERSIZE : INTEGER; -- Accessed Var of type: INTEGER

   next_out : INTEGER; -- Accessed Var of type: INTEGER

   Create is
         do
            c_create ("TAKE");
         end; -- Create

   pre_method is
      local
         p:OBJECT_PORT;
         cs : CODED_STRING;
         co : OBJECT_PORT;
      do
         cs.Create;
         cs.set_give_me("n");
         n := master_connection.send_msg(cs).the_integer;
         cs.set_give_me("BUFFERSIZE");
         BUFFERSIZE := master_connection.send_msg(cs).the_integer;
         cs.set_give_me("next_out");
         next_out := master_connection.send_msg(cs).the_integer;
         cs.set_reply("Early Return");
         master_connection.reply_msg(cs);
      end; -- pre_method


   post_method is
      local
         hhhh:CODED_STRING;
      do
         hhhh.Create;
         hhhh.set_integer(n);
         local_code.set_object_res("n",hhhh);
         local_code.com_send(master_connection);
         hhhh.Create;
         hhhh.set_integer(BUFFERSIZE);
         local_code.set_object_res("BUFFERSIZE",hhhh);
         local_code.com_send(master_connection);
         hhhh.Create;
         hhhh.set_integer(next_out);
         local_code.set_object_res("next_out",hhhh);
         local_code.com_send(master_connection);
       end; -- post_method

    do_method  is
          -- remove next_out item
 local
       do
          next_out := next_out + 1;
          if next_out = BUFFERSIZE + 1 then
             next_out := 1;
          end;  -- if

          n := n - 1;
```

```
        end

end; -- class TAKE
```

Appendix D

Example 3: Dining Philosophers

This appendix contains example code generated by the compiler produced for this thesis and hence is the parallelised-code versions of the classes describing the producer-consumer problem in section 11.5.

D.1 DINING class

D.1.1 DINING

```
class DINING
inherit
   COM
      rename process_command as c_process_command
,
      Create as c_create
      redefine process_command, pre_loop, method_connect;


feature

      phil1, phil2, phil3, phil4, phil5: COM_CONNECTION;

      f1, f2, f3, f4, f5: COM_CONNECTION;

   create is
      do
         c_create("DINING");
      end; -- create

   pre_loop is
      local
         cstr : CODED_STRING
      do
         cstr.Create;
         r_object_gen.new_method("mk_dining_METHOD_COURIER",host_name,
         0,"mk_dining", "mk_dining_METHOD");
         r_object_gen.new_method("live_METHOD_COURIER",host_name,
         0,"live", "live_METHOD");
      end; -- pre_loop
```

```
   method_connect(cmd:CODED_STRING;conn:COM_CONNECTION) is
      local
         str:STRING;
         cs : CODED_STRING;
         ms : MESSAGE;
      do
         str:=cmd.the_name;
         if str.equal("mk_dining_METHOD") then
            mk_dining := conn;
            ms.Create;
            cs.Create;
            cs.set_name(my_name);  -- courier required!
            cs.com_reply(conn);
            ms.set_method_name("mk_dining");
            cs.set_message(ms);
            mk_dining.send_command(cs);  -- as opposed to blocking send_msg
         elsif str.equal("live_METHOD") then
            live := conn;
         else
            info_display.display_msg(my_name,conn.talker_name,cmd,"UNKNOWN MESSAGE");
         end; -- if
      end; -- method_connect

   mk_dining : COM_CONNECTION;   --- METHOD; -- Create


   live : COM_CONNECTION;   --- METHOD;

   process_command(cmd : like last_command, conn : COM_CONNECTION) is
      do
         c_process_command(cmd, conn);
      end; -- process_command

end;  -- class DINING
```

D.1.2 LIVE

```
class LIVE export
repeat COMMAND
inherit COMMAND
rename Create as c_create
   redefine do_method, pre_method, post_method
feature

   f1 : COM_CONNECTION; -- Accessed Var of type: FORK

   f2 : COM_CONNECTION; -- Accessed Var of type: FORK

   f3 : COM_CONNECTION; -- Accessed Var of type: FORK

   phil1 : COM_CONNECTION; -- Accessed Var of type: PHILOSOPHER

   phil2 : COM_CONNECTION; -- Accessed Var of type: PHILOSOPHER

   phil3 : COM_CONNECTION; -- Accessed Var of type: PHILOSOPHER

   phil4 : COM_CONNECTION; -- Accessed Var of type: PHILOSOPHER

   phil5 : COM_CONNECTION; -- Accessed Var of type: PHILOSOPHER

   f4 : COM_CONNECTION; -- Accessed Var of type: FORK

   f5 : COM_CONNECTION; -- Accessed Var of type: FORK

   Create is
```

```
      do
         c_create ("LIVE");
      end; -- Create

pre_method is
   local
      p:OBJECT_PORT;
      cs : CODED_STRING;
      co : OBJECT_PORT;
   do
      cs.Create;
      cs.set_give_me("f1");
      co := master_connection.send_msg(cs).the_com; -- FORK
      if co.port = 0 then
         -- not created yet
      elsif co.port = f1.com_port.port and co.host.equal(f1.com_port.host) then
      -- no need to connect as we already have this object
      else
         connect_to_port(co.the_com.host,co.the_com.port);
         f1 := new_connection;
      end; -- if
      cs.set_give_me("f2");
      co := master_connection.send_msg(cs).the_com; -- FORK
      if co.port = 0 then
         -- not created yet
      elsif co.port = f2.com_port.port and co.host.equal(f2.com_port.host) then
      -- no need to connect as we already have this object
      else
         connect_to_port(co.the_com.host,co.the_com.port);
         f2 := new_connection;
      end; -- if
      cs.set_give_me("f3");
      co := master_connection.send_msg(cs).the_com; -- FORK
      if co.port = 0 then
         -- not created yet
      elsif co.port = f3.com_port.port and co.host.equal(f3.com_port.host) then
      -- no need to connect as we already have this object
      else
         connect_to_port(co.the_com.host,co.the_com.port);
         f3 := new_connection;
      end; -- if
      cs.set_give_me("phil1");
      co := master_connection.send_msg(cs).the_com; -- PHILOSOPHER
      if co.port = 0 then
         -- not created yet
      elsif co.port = phil1.com_port.port and co.host.equal(phil1.com_port.host) then
      -- no need to connect as we already have this object
      else
         connect_to_port(co.the_com.host,co.the_com.port);
         phil1 := new_connection;
      end; -- if
      cs.set_give_me("phil2");
      co := master_connection.send_msg(cs).the_com; -- PHILOSOPHER
      if co.port = 0 then
         -- not created yet
      elsif co.port = phil2.com_port.port and co.host.equal(phil2.com_port.host) then
      -- no need to connect as we already have this object
      else
         connect_to_port(co.the_com.host,co.the_com.port);
         phil2 := new_connection;
      end; -- if
      cs.set_give_me("phil3");
      co := master_connection.send_msg(cs).the_com; -- PHILOSOPHER
      if co.port = 0 then
         -- not created yet
      elsif co.port = phil3.com_port.port and co.host.equal(phil3.com_port.host) then
      -- no need to connect as we already have this object
      else
```

```
      connect_to_port(co.the_com.host,co.the_com.port);
      phil3 := new_connection;
    end; -- if
    cs.set_give_me("phil4");
    co := master_connection.send_msg(cs).the_com; -- PHILOSOPHER
    if co.port = 0 then
      -- not created yet
    elsif co.port = phil4.com_port.port and co.host.equal(phil4.com_port.host) then
    -- no need to connect as we already have this object
    else
      connect_to_port(co.the_com.host,co.the_com.port);
      phil4 := new_connection;
    end; -- if
    cs.set_give_me("phil5");
    co := master_connection.send_msg(cs).the_com; -- PHILOSOPHER
    if co.port = 0 then
      -- not created yet
    elsif co.port = phil5.com_port.port and co.host.equal(phil5.com_port.host) then
    -- no need to connect as we already have this object
    else
      connect_to_port(co.the_com.host,co.the_com.port);
      phil5 := new_connection;
    end; -- if
    cs.set_give_me("f4");
    co := master_connection.send_msg(cs).the_com; -- FORK
    if co.port = 0 then
      -- not created yet
    elsif co.port = f4.com_port.port and co.host.equal(f4.com_port.host) then
    -- no need to connect as we already have this object
    else
      connect_to_port(co.the_com.host,co.the_com.port);
      f4 := new_connection;
    end; -- if
    cs.set_give_me("f5");
    co := master_connection.send_msg(cs).the_com; -- FORK
    if co.port = 0 then
      -- not created yet
    elsif co.port = f5.com_port.port and co.host.equal(f5.com_port.host) then
    -- no need to connect as we already have this object
    else
      connect_to_port(co.the_com.host,co.the_com.port);
      f5 := new_connection;
    end; -- if
    cs.set_reply("Early Return");
    master_connection.reply_msg(cs);
  end; -- pre_method


post_method is
  local
    hhhh:CODED_STRING;
  do
    local_code.set_object_res("f1",f1.remote_port); -- FORK
    local_code.com_send(master_connection);
    local_code.set_object_res("f2",f2.remote_port); -- FORK
    local_code.com_send(master_connection);
    local_code.set_object_res("f3",f3.remote_port); -- FORK
    local_code.com_send(master_connection);
    local_code.set_object_res("phil1",phil1.remote_port); -- PHILOSOPHER
    local_code.com_send(master_connection);
    local_code.set_object_res("phil2",phil2.remote_port); -- PHILOSOPHER
    local_code.com_send(master_connection);
    local_code.set_object_res("phil3",phil3.remote_port); -- PHILOSOPHER
    local_code.com_send(master_connection);
    local_code.set_object_res("phil4",phil4.remote_port); -- PHILOSOPHER
    local_code.com_send(master_connection);
    local_code.set_object_res("phil5",phil5.remote_port); -- PHILOSOPHER
    local_code.com_send(master_connection);
```

```
      local_code.set_object_res("f4",f4.remote_port); -- FORK
      local_code.com_send(master_connection);
      local_code.set_object_res("f5",f5.remote_port); -- FORK
      local_code.com_send(master_connection);
   end; -- post_method

do_method  is
   local
   -- Generated locals
      l0 : COM_CONNECTION;
      l1 : COM_CONNECTION;
      l2 : COM_CONNECTION;
      l3 : COM_CONNECTION;
      l4 : COM_CONNECTION;
      l5 : COM_CONNECTION;
      l6 : COM_CONNECTION;
      l7 : COM_CONNECTION;
      l8 : COM_CONNECTION;
      l9 : COM_CONNECTION;

   do
      from
      until
         False
      loop
         local_comp.clear;
         local_msg.clear;
         local_msg.set_method_name("hungry");
         local_comp.extend(local_msg.coded_string);
         if phil1.send_msg(local_comp.coded_string).the_boolean then
            local_comp.clear;
            local_msg.clear;
            local_comp.clear;
            local_msg.clear;
            l0 := f1; -- message value generated (break on a non-COM)
            local_comp.clear;
            local_msg.clear;
            l1 := f2; -- message value generated (break on a non-COM)
            local_msg.set_method_name("eat");
            local_code.set_com(l0.com_port);--FORK
            local_msg.extend(local_code);
            local_code.set_com(l1.com_port);--FORK
            local_msg.extend(local_code);
            local_comp.extend(local_msg.coded_string);
            phil1.send_command(local_comp.coded_string);
         else
            local_comp.clear;
            local_msg.clear;
            local_msg.set_method_name("think");
            local_comp.extend(local_msg.coded_string);
            phil1.send_command(local_comp.coded_string);
         end;  -- if

         local_comp.clear;
         local_msg.clear;
         local_msg.set_method_name("hungry");
         local_comp.extend(local_msg.coded_string);
         if phil2.send_msg(local_comp.coded_string).the_boolean then
            local_comp.clear;
            local_msg.clear;
            local_comp.clear;
            local_msg.clear;
            l2 := f2; -- message value generated (break on a non-COM)
            local_comp.clear;
            local_msg.clear;
            l3 := f3; -- message value generated (break on a non-COM)
            local_msg.set_method_name("eat");
            local_code.set_com(l2.com_port);--FORK
```

```
   local_msg.extend(local_code);
   local_code.set_com(l3.com_port);--FORK
   local_msg.extend(local_code);
   local_comp.extend(local_msg.coded_string);
   phil2.send_command(local_comp.coded_string);
else
   local_comp.clear;
   local_msg.clear;
   local_msg.set_method_name("think");
   local_comp.extend(local_msg.coded_string);
   phil2.send_command(local_comp.coded_string);
end;  -- if

local_comp.clear;
local_msg.clear;
local_msg.set_method_name("hungry");
local_comp.extend(local_msg.coded_string);
if phil3.send_msg(local_comp.coded_string).the_boolean then
   local_comp.clear;
   local_msg.clear;
   local_comp.clear;
   local_msg.clear;
   l4 := f3; -- message value generated (break on a non-COM)
   local_comp.clear;
   local_msg.clear;
   l5 := f4; -- message value generated (break on a non-COM)
   local_msg.set_method_name("eat");
   local_code.set_com(l4.com_port);--FORK
   local_msg.extend(local_code);
   local_code.set_com(l5.com_port);--FORK
   local_msg.extend(local_code);
   local_comp.extend(local_msg.coded_string);
   phil3.send_command(local_comp.coded_string);
else
   local_comp.clear;
   local_msg.clear;
   local_msg.set_method_name("think");
   local_comp.extend(local_msg.coded_string);
   phil3.send_command(local_comp.coded_string);
end;  -- if

local_comp.clear;
local_msg.clear;
local_msg.set_method_name("hungry");
local_comp.extend(local_msg.coded_string);
if phil4.send_msg(local_comp.coded_string).the_boolean then
   local_comp.clear;
   local_msg.clear;
   local_comp.clear;
   local_msg.clear;
   l6 := f4; -- message value generated (break on a non-COM)
   local_comp.clear;
   local_msg.clear;
   l7 := f5; -- message value generated (break on a non-COM)
   local_msg.set_method_name("eat");
   local_code.set_com(l6.com_port);--FORK
   local_msg.extend(local_code);
   local_code.set_com(l7.com_port);--FORK
   local_msg.extend(local_code);
   local_comp.extend(local_msg.coded_string);
   phil4.send_command(local_comp.coded_string);
else
   local_comp.clear;
   local_msg.clear;
   local_msg.set_method_name("think");
   local_comp.extend(local_msg.coded_string);
   phil4.send_command(local_comp.coded_string);
end;  -- if
```

```
            local_comp.clear;
            local_msg.clear;
            local_msg.set_method_name("hungry");
            local_comp.extend(local_msg.coded_string);
            if phil5.send_msg(local_comp.coded_string).the_boolean then
               local_comp.clear;
               local_msg.clear;
               local_comp.clear;
               local_msg.clear;
               l8 := f5; -- message value generated (break on a non-COM)
               local_comp.clear;
               local_msg.clear;
               l9 := f1; -- message value generated (break on a non-COM)
               local_msg.set_method_name("eat");
               local_code.set_com(l8.com_port);--FORK
               local_msg.extend(local_code);
               local_code.set_com(l9.com_port);--FORK
               local_msg.extend(local_code);
               local_comp.extend(local_msg.coded_string);
               phil5.send_command(local_comp.coded_string);
            else
               local_comp.clear;
               local_msg.clear;
               local_msg.set_method_name("think");
               local_comp.extend(local_msg.coded_string);
               phil5.send_command(local_comp.coded_string);
            end;;
         end;  -- loop

      end

end; -- class LIVE
```

D.2 FORK class

D.2.1 PICKUP

```
class PICKUP export
repeat COMMAND
inherit COMMAND
rename Create as c_create
   redefine do_method, pre_method, post_method
feature

   Create is
         do
            c_create ("PICKUP");
         end; -- Create

   pre_method is
      local
         p:OBJECT_PORT;
         cs : CODED_STRING;
         co : OBJECT_PORT;
      do
         cs.Create;
         cs.set_reply("Early Return");
         master_connection.reply_msg(cs);
      end; -- pre_method


   post_method is
```

```
      local
         hhhh:CODED_STRING;
      do
      end; -- post_method

   do_method  is
      local
      do -- code to pickup fork

      end

end; -- class PICKUP
```

D.2.2 USE

```
class USE export
repeat COMMAND
inherit COMMAND
rename Create as c_create
   redefine do_method, pre_method, post_method
feature

   Create is
         do
            c_create ("USE");
         end; -- Create

   pre_method is
      local
         p:OBJECT_PORT;
         cs : CODED_STRING;
         co : OBJECT_PORT;
      do
         cs.Create;
         cs.set_reply("Early Return");
         master_connection.reply_msg(cs);
      end; -- pre_method


   post_method is
      local
         hhhh:CODED_STRING;
      do
      end; -- post_method

   do_method  is
      local
      do -- code to actually use the fork

      end

end; -- class USE
```

D.2.3 PUTDOWN

```
class PUTDOWN export
repeat COMMAND
inherit COMMAND
rename Create as c_create
   redefine do_method, pre_method, post_method
feature
```

```
   Create is
         do
            c_create ("PUTDOWN");
         end; -- Create

   pre_method is
      local
         p:OBJECT_PORT;
         cs : CODED_STRING;
         co : OBJECT_PORT;
      do
         cs.Create;
         cs.set_reply("Early Return");
         master_connection.reply_msg(cs);
      end; -- pre_method


   post_method is
      local
         hhhh:CODED_STRING;
      do
      end; -- post_method

   do_method  is
      local
      do -- code to putdown fork

      end

end; -- class PUTDOWN
```

D.3 PHILOSOPHER class

D.3.1 PHILOSOPHER

```
class PHILOSOPHER export

   repeat COM, eat,think,hungry
inherit
   COM
      rename process_command as c_process_command
,
      Create as c_create
      redefine process_command, pre_loop, method_connect;


feature

      hungry: BOOLEAN;

   create is
      do
         c_create("PHILOSOPHER");
      end; -- create

   pre_loop is
      local
         cstr : CODED_STRING
      do
         cstr.Create;
         r_object_gen.new_method("think_METHOD_COURIER",host_name,
         0,"think", "think_METHOD");
```

```
      r_object_gen.new_method("eat_METHOD_COURIER",host_name,
      0,"eat", "eat_METHOD");
   end; -- pre_loop

method_connect(cmd:CODED_STRING;conn:COM_CONNECTION) is
   local
      str:STRING;
      cs : CODED_STRING;
      ms : MESSAGE;
   do
      str:=cmd.the_name;
      if str.equal("think_METHOD") then
         think := conn;
      elsif str.equal("eat_METHOD") then
         eat := conn;
      else
         info_display.display_msg(my_name,conn.talker_name,cmd,"UNKNOWN MESSAGE");
      end; -- if
   end; -- method_connect

eat : COM_CONNECTION;   --- METHOD; -- eat


think : COM_CONNECTION;   --- METHOD;

process_command(cmd : like last_command, conn : COM_CONNECTION) is
   local
      local_msg : MESSAGE;
      cstr : CODED_STRING;

   do
      cstr.Create;
      if cmd.is_a_reply then
      elsif cmd.is_a_message then
         local_msg ?= cmd.the_message;
         if local_msg.void then
            io.putstring("ERR:HELLPPPP!")
         end; -- if
         if not check_access(conn.talker_name) then
         -- check if this caller has access rights
               queue_request(cmd,conn);
         elsif local_msg.method_name.equal("think") then
            cmd.com_send(think); -- command
         elsif local_msg.method_name.equal("hungry") then
            cstr.set_boolean(hungry);
            cstr.com_reply(conn);
         elsif local_msg.method_name.equal("eat") then
            cmd.com_send(eat);   -- command
         else
            io.putstring("\nUNDEFINED MESSAGE IN AN INSTANCE OF");
            io.putstring(class_name);
            io.putstring("\n MESSAGE IS:");
            io.putstring(cmd.string_rep);
         end; -- if
      else
         c_process_command(cmd,conn);
      end; -- if
   end; -- process_command

end;  -- class PHILOSOPHER
```

D.3.2 EAT

```
class EAT export
repeat COMMAND
```

```
inherit COMMAND
rename Create as c_create
   redefine do_method, pre_method, post_method
feature

   hungry : BOOLEAN; -- Accessed Var of type: BOOLEAN

   f1 : COM_CONNECTION;

   f2 : COM_CONNECTION;

   Create is
         do
            c_create ("EAT");
         end; -- Create

   pre_method is
      local
         p:OBJECT_PORT;
         cs : CODED_STRING;
         co : OBJECT_PORT;
      do
         cs.Create;
         p := msg.i_th(1).the_com;
         connect_to_port(p.host,p.port);
         f1 := new_connection;
         f1.set_object_port(p.deep_clone);
         p := msg.i_th(2).the_com;
         connect_to_port(p.host,p.port);
         f2 := new_connection;
         f2.set_object_port(p.deep_clone);
         cs.set_give_me("hungry");
         hungry := master_connection.send_msg(cs).the_boolean;
         cs.set_reply("Early Return");
         master_connection.reply_msg(cs);
      end; -- pre_method


   post_method is
      local
         hhhh:CODED_STRING;
      do
         hhhh.Create;
         hhhh.set_boolean(hungry);
         local_code.set_object_res("hungry",hhhh);
         local_code.com_send(master_connection);
      end; -- post_method

   do_method  is

      local
         ate: INTEGER;
      do
         local_comp.clear;
         local_msg.clear;
         local_msg.set_method_name("pickup");
         local_comp.extend(local_msg.coded_string);
         f1.send_command(local_comp.coded_string);
         local_comp.clear;
         local_msg.clear;
         local_msg.set_method_name("pickup");
         local_comp.extend(local_msg.coded_string);
         f2.send_command(local_comp.coded_string);
         from
         until
            not hungry
         loop  -- do some eating using forks
```

```
            local_comp.clear;
            local_msg.clear;
            local_msg.set_method_name("use");
            local_comp.extend(local_msg.coded_string);
            f1.send_command(local_comp.coded_string);
            local_comp.clear;
            local_msg.clear;
            local_msg.set_method_name("use");
            local_comp.extend(local_msg.coded_string);
            f2.send_command(local_comp.coded_string);
            ate := ate + 1;   -- set hungry to False when full, e.g.

            if ate > 10000 then
               hungry := False;
            end;;
         end;  -- loop

         local_comp.clear;
         local_msg.clear;
         local_msg.set_method_name("putdown");
         local_comp.extend(local_msg.coded_string);
         f2.send_command(local_comp.coded_string);
         local_comp.clear;
         local_msg.clear;
         local_msg.set_method_name("putdown");
         local_comp.extend(local_msg.coded_string);
         f1.send_command(local_comp.coded_string);
      end

end; -- class EAT
```

D.3.3 THINK

```
class THINK export
repeat COMMAND
inherit COMMAND
rename Create as c_create
   redefine do_method, pre_method, post_method
feature

   hungry : BOOLEAN; -- Accessed Var of type: BOOLEAN

   Create is
         do
            c_create ("THINK");
         end; -- Create

   pre_method is
      local
         p:OBJECT_PORT;
         cs : CODED_STRING;
         co : OBJECT_PORT;
      do
         cs.Create;
         cs.set_give_me("hungry");
         hungry := master_connection.send_msg(cs).the_boolean;
         cs.set_reply("Early Return");
         master_connection.reply_msg(cs);
      end; -- pre_method


   post_method is
      local
         hhhh:CODED_STRING;
      do
```

```
            hhhh.Create;
            hhhh.set_boolean(hungry);
            local_code.set_object_res("hungry",hhhh);
            local_code.com_send(master_connection);
         end; -- post_method

      do_method  is

         local
            thunk: INTEGER;
         do
            from
            until
               hungry
            loop  -- do some thinking

               thunk := thunk + 1;  -- set hungry to True when hungry, e.g.

               if thunk > 10000 then
                  hungry := False;
               end;;
            end;  -- loop

         end

end; -- class THINK
```