

## Chapter 6

# Object Oriented Program Derivation

This chapter examines how implementations can be derived from the formal object oriented LOTOS designs which arise from application of the methods defined in chapters 2 through to 5.

- **6.1: High-level Object Oriented Design as Input to Implementation**

This section introduces implementation as an extension to design, and reviews a range of programming languages and environments which could be used to implement the object oriented LOTOS design specifications. It argues that, in general, executable languages can express three aspects of software specification: data structure, function (data transformation) and flow of control, and shows that different programming languages place different degrees of emphasis on each. With this in mind, the implementation of object oriented requirements using non-object oriented languages is first considered. Then, the advantages of working in an object oriented programming environment are put forward.

- **6.2: Object Oriented Programming: The Alternatives**

Section 6.2 examines the different types of object oriented programming languages (and environments) which are currently available. It begins by defining the four main roles of object oriented programmers: interfacing with designers, writing code, producing documentation and testing. Different characteristics of object oriented languages are identified and, based on these characteristics, a review of object oriented programming languages is given. Finally, Eiffel is chosen as the object oriented programming languages most suitable for implementing the formal object oriented LOTOS designs.

- **6.3: Translating Design To Implementation: Mapping Semantics**

This section begins by reviewing the concept of targetted design: informality, in programming language semantics, is argued to make the targetting process more complex, and the future development of a programming language with formal semantics (based on the O-LSTS functional model, and a process algebra communication model) is recommended. The informal semantics of object oriented programming languages are reluctantly accepted as a necessary evil at this stage of the research. The remote procedure call communication model (RPC) is put forward as the best option when targetting design towards an Eiffel implementation.

- **6.4: Producing Eiffel from Procedural Object Oriented LOTOS Designs**

Section 6.4 shows how Eiffel code can be developed from the formal object oriented designs, in order to meet the requirements. Initially, the implementation work of this thesis is placed within a set of reasonable bounds. It is not possible to examine all implementation issues and so restrictions are placed on the type of work which this thesis addresses. Then, an overview of the main implementation problem is given, namely matching value and reference semantics. The main body of this section gives a high-level view on the production of Eiffel code from formal designs: implementing object-based requirements, object-oriented requirements, using assertions and exceptions, and a potpourri of other relevant issues.

- **6.5: A Question Of Concurrency And Distribution**

This section concludes this chapter by examining how the formal object oriented development process can be targetted towards concurrent or distributed implementations. It begins by stating the obvious advantages of concurrency and distribution in software systems, whilst re-stating the reasons for concentrating on a sequential implementation approach in this thesis. Alternative views of the relative merits of combining object oriented and concurrent models are given. The main problem for concurrent object oriented languages is argued to be that of scale. Using our object oriented design method is shown to provide a solution to the problem of complexity explosion when mapping objects to processes. Then, references to the conflicting requirements of object oriented and concurrent semantics are given. This section concludes by stating that the formal object oriented development approach, as advocated in this thesis, has the potential for being used to construct concurrent implementations.

## 6.1 High-level Object Oriented Design as Input to Implementation

In chapter 5, the importance of targetting a design towards a particular implementation language (environment) is stressed. Provided this is done appropriately, coding should then be a natural extension of the design process. Writing code should, in theory, be almost mechanical in nature, since the designers have done all the hard work. However, matching design specification semantics with different implementation language semantics is not always a simple task.

There are three orthogonal aspects to object oriented LOTOS designs:

- The communication model, i.e. the semantics of message passing (service requests and service fulfilment).
- The composition structure.
- The subclassing hierarchy and associated static and dynamic classification properties.

Each of these aspects must be mapped onto the implementation language. It is the designer's role to make this mapping as simple as possible.

The implementation process is made easier when the design semantics are close to the programming language semantics. Chapter 5 defines the object oriented design semantics in a way which

separates the functional, compositional and classification properties (in the ADT part) from the interaction, control and communication properties (in the process algebra part). Clearly, some languages will be better suited to implementing the formal designs than others. However, since, in general, programming languages provide the same computational power, the object oriented concepts can be mapped into non-object oriented language constructs.

### 6.1.1 An Overview of Programming Languages and Implementation Concerns

In general, executable (programming) languages can express properties with regard to three different aspects of a software system: data structure, data transformation (function) and flow of control. (This 3-dimensional categorisation is first commented on in chapter 2, when different aspects of analysis are considered.) Formal object oriented analysis concentrates on data structure and data transformation, whilst providing a service-request semantics which can be used to form the basis of a wide range of communication and control-flow models. The process algebra part of the formal object oriented designs make more concrete the flow-of-control aspects of the proposed solution to the requirements. Consequently, for implementation to be straightforward, it is necessary that the chosen programming language is rich in expression with respect to data structure, data transformation and control flow (data communication).

#### 6.1.1.1 Data Structure and Data Relationships

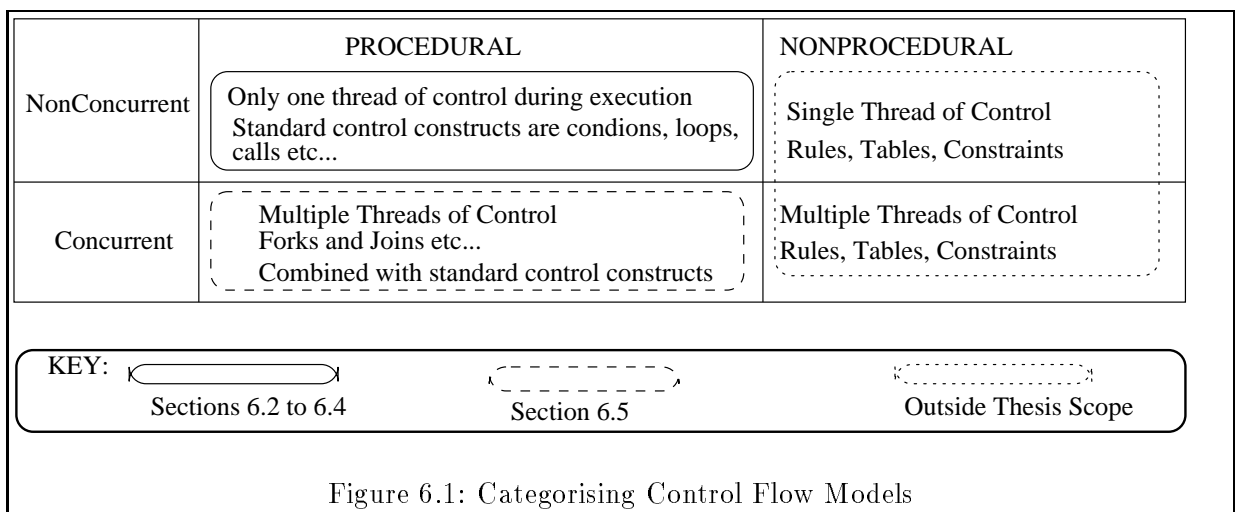
The declarative composition and subclassing relations are fundamental aspects of the object oriented formal models. Programming languages, in general, have a declarative (non procedural) element, used to define problem specific data structures. Most programming languages provide a means of defining new data structures as groups (commonly called records or structures) of already defined data structures. These mechanisms provide an obvious means of modelling composition. However, most programming languages do not provide a mechanism for defining subclassing-like relationships between data structures. In such cases, a subclassing model must be built on top of the declarative constructs, using composition in some conventional way. Such a work-around solution is made even more complex by the polymorphic requirements inherent in formal designs.

#### 6.1.1.2 Data Transformation (Function)

Data transformation is commonly provided by primitive operators (whose semantics are defined as part of the language), together with a means of constructing non-primitive operations (usually in the form of subprograms which can be called ‘as-if’ primitive). Procedural languages offer a similar degree of support for expressing data transformation properties. These transformation constructs provide a natural means of modelling the object oriented notion of attributes/methods. Non-procedural languages provide a different challenge to object oriented modellers.

### 6.1.1.3 Flow of Control

The different models of control flow evident in programming languages are categorised in figure 6.1. The main body of this chapter, namely sections 6.2 to 6.4, concentrates on non-concurrent implementation models. Section 6.5 examines the suitability of our formal object oriented development strategy with respect to the production of a concurrent implementation.



### 6.1.2 Implementation Outside an Object Oriented Framework

All languages (including object oriented programming languages) vary in their ability to support object oriented concepts. In particular, they differ in their ability to support the primitive concepts in the formal designs. Programming languages represent a compromise between: achieving a conceptual framework of understanding, being efficient and offering compatibility with other systems (and languages). Achieving a balance between these three requirements is, principally, what tempers the programming language semantics. Implementation is the process of matching these programming language semantics to the given design semantics. Using a non-object oriented language to implement object oriented requirements needs great care since there is no direct support, from the implementation environment, in maintaining the object oriented properties. However, it can be done. For example, Eiffel [84] is compiled into C, and the resulting C adheres to syntactic conventions which give it an object oriented flavour. By directly following these conventions, it is possible to produce a C implementation without using Eiffel. This implementation approach is very difficult without the type of support that the Eiffel programming environment provides. To give a flavour of how LOTOS object oriented designs can be implemented in non-object oriented programming languages, implementation in three different environments is considered:

- Using a purely functional programming language.
- Using an imperative programming language.
- Using a relational database language.

### 6.1.2.1 Functional Implementation

Functional and object oriented languages appear, at first glance, to have much in common: both place emphasis on the the notion of categorisation in the form of type and class, respectively. Much debate has arisen concerning the differences and similarities between the notions of type and class (and subtyping and subclassing). Section 2.5 provides an overview of the discussion and puts forward the stance of this thesis.

Given an ADT specification of object oriented requirements, as generated by the object oriented analysis, it is clear that it should be possible to directly implement it in a functional language: a simple mapping between sort and type, and operation and function would form the basis of such an implementation. However, as with the ACT ONE analysis model, a functional language which does not provide polymorphism will require additional work on the part of the programmer (or code generator) to fulfil the polymorphic requirements. Some functional programming languages provide polymorphic types, e.g. Miranda [111], but such polymorphism is ad-hoc rather than constrained. The work by Wadler and Blott [122] reviews the problems introduced by ad hoc polymorphism in functional languages. The research language Haskell [4] is an attempt to introduce object oriented properties into a purely functional programming framework, but this work is still incomplete. Perhaps the most interesting work in combining object oriented semantics with functional semantics is embodied in FOOPS (Functional Object Oriented Programming System) [60]. Unfortunately, the primitive definitions in the object oriented semantic framework used in this thesis do not correspond directly to the FOOPS notions. As such, although the mapping between the O-LSTS semantics and a functional semantics is an interesting task, a functionally based implementation was not carried out as part of this work.

### 6.1.2.2 Imperative Implementation

The expressive similarities of different imperative programming languages can be taken advantage of in the definition of a general algorithm for the imperative implementation of object oriented requirements. The key stages to such an algorithm are:

- **Stage 1: Model classes as data structures.**

Classes are specified in the formal designs as structured processes. The composition structure is either: explicit in the decomposition of the process into a set of component processes, or implicit in the ACT ONE sort which parameterises the behaviour of class instances. In both cases, this structure can be directly translated into a record structure in an imperative implementation language. The fields of the record correspond to the components of the class. Variable record mechanisms (whether provided as primitive language constructs or defined by the programmer) can be used to model classes with different structures. Class literals can be simply implemented as enumerated types.

- **Stage 2: Model subclassing in data structures.**

One approach to modelling subclassing is to flatten the class hierarchy structure: all the code for each attribute of a class is then defined in the particular class body. This can lead to

multiple copies of the same code, although in many imperative languages this duplication can be controlled using meta-language constructs like macros. A second approach is to model an inheritance facility using pointers to code bodies (which are shared among classes with common roots in the subclassing hierarchy). This second approach can be extended to model the dynamic binding of a service request to code at run-time. A third approach is to translate subclassing relationships into delegation relationships. Instead of a group of subclasses ‘inheriting’ code from a common superclass, the superclass behaviour being inherited can be defined in an extra component common to each of the subclasses. In this way, ‘inherited’ behaviour is provided by delegation. The programmers must choose which approach is best suited to their particular language.

- **Stage 3: Model polymorphism in data structures.**

In our object oriented model, an object can be treated as if it was a member of any of its superclasses. This is polymorphism: the object is dynamically bound<sup>1</sup> to a particular class at run-time. The polymorphism must be controlled so that an object is only ever re-bound to a superclass of its current class. Polymorphism makes type checking complex. In imperative implementation languages, where the static type checking is comprehensive, it is necessary to model polymorphism using operation overloading and coercion. In languages with weak static typing, often the compiler does not check that typing properties are met, and so polymorphism is ad-hoc. This type of language can be used for the implementation of object oriented requirements, provided the typing requirements have been statically checked outside the domain of the programming language semantics.

- **Stage 4: Model attributes as functions.**

Every attribute must be defined to have at least one argument, the implicit notion of *self*, i.e. the object being asked to service the attribute request. It is useful to define a convention that this argument is always the first one in the list of attribute operation parameters. Implementers must decide whether the other parameters should be passed by value or by reference. When passing parameter values as references, there may be side-effects if accessor or dual attributes are requested of the parameter. Consequently, for safety, it is better to pass the arguments as values. However, for efficiency reasons, it is often better to pass parameter values as references.

A second concern when defining functions is the way in which they are named. Different programming languages have different syntactic restrictions placed on the naming of identifiers. It is important that a naming convention is found which, within these restrictions, can be used coherently and consistently. For example, a unique identity for each function can usually be generated by combining the class and attribute names in an appropriate fashion.

- **Stage 5: Model creation/initialisation routines.**

Creating a system corresponds to instantiating a member of a class. Classes can be either:

- **Purely static.**

When a purely static class instance is created, the resulting system has a persistent struc-

---

<sup>1</sup>This is different from the notion of dynamically binding a service to code.

ture, i.e. servicing an arbitrary sequence of external attribute requests does not change the structured representation of the initial object. Further, in a purely static class, the components of the class are required to be purely static instances.

- **Impurely static.**

As with a purely static class, the instances have persistent structure but, in the impure case, class components are not required to be purely static.

- **Dynamic.**

A dynamic class instance, defined as a particular set of component objects, may be transformed into a different set of components by fulfilling some sequence of service requests.

Except in the purely static case, it will be necessary to be able to create and destroy system component objects (subsystems) during the lifetime of the system. These subsystems can, when the use of references is strictly controlled, be implemented as stack-based variables. The imperative language compiler can then automatically cope with memory allocation and deallocation.

- **Stage 6: Model encapsulation.**

Object oriented semantics require encapsulation of an object so that access to its state must be done through its external interface. This requirement is not standard in imperative programming languages. Module-like constructs provide encapsulation in some languages, but there are difficulties in defining a correspondence between objects and modules, especially in dynamically structured systems. Furthermore, confusion can arise when systems have multiple instances of the same module. It is better to enforce a convention that all access to structured data (in record form) must be through the external attribute functions.

- **Stage 7: Model concurrency, or lack of it.**

The object oriented LOTOS design models are easier to implement if the concurrency (modelled using the parallel operators) is removed. In this case, message passing (service request/service response) events can be modelled imperatively using remote procedure calls. Section 6.5 examines the issues which arise when the final design has concurrent aspects which are intended to be carried through to concurrent implementation language constructs.

These seven stages provide the basis for implementing object oriented requirements in an imperative language. Of course, this mapping of object oriented requirements is not the whole story for the implementers. They must also consider coding a user interface, fulfilling non-functional requirements, documentation, testing, etc . . . (see 6.2). However, these aspects can be done *as-if* the implementation language was object oriented, provided the seven stages above are complete.

### 6.1.2.3 Implementation Using a Relational DataBase

Before examining implementation using object oriented programming languages, a final, less obvious, alternative is briefly considered: using a relational database.

When the object oriented requirements place emphasis on the persistence of data, i.e. data that exists beyond the lifetime of a single program execution, then a permanent data store is required.

Further, often the functional requirements of such systems are concerned with information retrieval and update: controlled access to different fields of data in a large data structure. Given such a requirements model, it is natural to think of implementing the system as a data base. Fortunately, high-level database languages exist to provide the core behavioural framework onto which particular data structure, and associated functionality, can be built. Relational database languages (see [23, 74] for example) have the potential to provide a sound basis on which to implement complex software systems of persistent objects.

### 6.1.3 Implementation in an Object Oriented Environment: The Advantages

Computational power is not an issue when choosing one implementation language over another, since programming languages can, in general compute whatever is computable. Section 6.1.2 gives an overview of how object oriented constructs can be modelled (and therefore implemented) in non-object oriented programming languages. In such an approach, the imperative language is used to construct a model of the object oriented semantics. There are inherent difficulties when implementing on top of such a model:

- The mapping between the object oriented primitives, in the LOTOS designs, and the final implementation language primitives is more complex than necessary.
- The imperative language does not provide error protection facilities, in the form of type checking, which can automatically check the object oriented implementation to guarantee it fulfils the complex correctness properties associated with a polymorphic language.
- The testing of the implementation becomes more complex since it is necessary to test both the functional requirements and the correct modelling of object oriented primitives.

One of the advantages of using an object oriented programming language is the consistent framework of conceptualisation between analysis, design and implementation<sup>2</sup>. Unfortunately, although the primitive concepts are common, the underlying semantics of the primitive concepts is not standard. Consequently, there is still a need to model the object oriented requirements primitives, as specified in our LOTOS designs, onto an object oriented implementation language. However, in most cases, the object oriented programming language semantics are closer to our design semantics than for non-object oriented programming languages, and the mapping is therefore much simpler. In particular, many of the mapping steps needed for imperative implementation (see 6.1.2.2) are unnecessary when using an object oriented programming language.

## 6.2 Object Oriented Programming (OOP): The Alternatives

### 6.2.1 The Roles of Object Oriented Programmers

Object oriented programmers have four main roles: interfacing with designers, coding, documenting and testing.

---

<sup>2</sup>Chapter 2 examines all the advantages of working in an object oriented framework, many of which are related to the conceptual consistency.

### 6.2.1.1 Interfacing With Designers

The role of designers is to target the requirements towards a particular implementation architecture. This targetting is at three different levels:

- Matching the object oriented semantics in the design with the programming language semantics, particularly with regard to the dynamic classification and communication properties.
- Matching the compositional structure of the design to resources in the implementation language. In particular, this means re-using already coded design/implementation components.
- Ensuring that non-functional requirements can be met by the chosen implementation environment.

It is advised that implementers help designers to make appropriate design decisions.

### 6.2.1.2 Coding

After object oriented analysis and design, the implementers may still have much to do:

- Code new classes and make these available for re-use (in some sort of package facility).
- Place new design classes into the class hierarchy, if not already done during design.
- Identify new generic classes and define these for re-use.
- Fulfil the non-functional requirements.
- Match the static and dynamic typing requirements to the programming language.
- Provide a user interface to the system: define a means of representing system state, a means of dynamically interacting with the system, and a way of storing and retrieving previous systems.
- Resolve the unspecified behaviour associated with exceptions, which was not dealt with during design.
- Resolve implementation freedom.

These tasks are clearly inter-related in a complex way. This thesis is not an examination of object oriented programming techniques and as such we do not examine the programming process in great detail.

### 6.2.1.3 Document The Implementation With Respect To Design

The formal design forms the basis of the code documentation. Each implementation class has an associated design component. The OO ACT ONE specification (defined by the ACT ONE code) for each sort acts as a good statement of functional and structural properties. As such, we recommend that it be included in the code (in the form of a comment). The process algebra specification of the communication model can be included when its requirements are complex: when a consistent RPC model is enforced it is not necessary to include the communication information.

Another important role of the documentation is to comment on differences between design and implementation. For example, when sharing is used for efficiency, or concrete state does not match

abstract state. Further, documentation must deal with re-use issues: where predefined code has come from, and where re-usable components are stored for future use. Documentation must also deal with testing and user manuals<sup>3</sup>.

#### 6.2.1.4 Testing

The formal approach advocated in this thesis cannot guarantee that the resulting implementation provides the required behaviour since there are two informal steps: customer communication of requirements<sup>4</sup> and implementation. Implementation is said to be informal because there is no formal mapping between the semantics of the LOTOS designs and the semantics of the resulting executable code. However, the formal analysis and design stages do guarantee that the requirements model is fulfilled by the final design, and this final design is unambiguous. Further, the object oriented framework aids understanding of these formal models. As such, it helps to cover the informality gap at each end of the development process. Testing is the process by which implementers bridge the gaps at their end of development.

Code is tested against the final object oriented design (the initial requirements, implicit in the design, have already been validated by the customer). The structure in the design matches structure in the implementation, to a great extent, and consequently the testing process can be incremental.

### 6.2.2 Characterisation of OOP Languages

Object oriented programming languages vary in their support of object oriented concepts. The object oriented semantic framework, defined in chapter 3, is the basis upon which we evaluate the suitability of languages for implementing formal object oriented designs in LOTOS. Object oriented characteristics are categorised into three groups:

- **Essential**

These characteristics are the minimum requirements for a language to be considered suitable for implementation.

- **Important**

It is important that these characteristics are evident in the chosen programming language if the formal object oriented development method is to progress past the research stage, and gain initial acceptance in industry.

- **Beneficial**

Beneficial characteristics are those which could eventually positively influence the widescale adoption of formal object oriented development within industry.

It is not always clear whether or not an implementation language (or environment) exhibits a particular characteristic. When a distinction is necessary, the following categorisation is useful:

---

<sup>3</sup>It is beyond the scope of this thesis to examine the production of customer documentation.

<sup>4</sup>It is common for the customer to validate the requirements model as being correct even when it does not exactly represent their needs. In an ideal environment, the formal requirements act as a contract between customer and software developer so that there is a level of customer liability.

- **Directly Supported**

Directly supported characteristics are provided by the language primitives, but not necessarily enforced.

- **Supported**

Supported characteristics are provided as elements of the language libraries, or can be easily coded as such.

- **Unsupported**

Unsupported characteristics can be modelled by the language, but there is no language or library support.

### 6.2.2.1 Essential Characteristics

We regard the following characteristics to be essential in a programming language which is to be used for implementing the object oriented LOTOS designs:

- **Classification**

All object oriented languages provide a means of defining classes of behaviour. In most cases, these classes are defined to have state attributes<sup>5</sup>. Objects are references to particular instances of a class, in which the state attributes have been set to particular values.

- **Encapsulation**

Object state must be encapsulated behind an interface. In some languages, the state attributes cannot be accessed directly. In others, attributes must be declared *private* if direct access is to be prohibited. Unfortunately, some languages facilitate the declaration of state attributes as *private*, but do not enforce the privacy (see Smalltalk [58, 57], for example).

- **Composition**

All object oriented languages facilitate a form of composition, usually by allowing state attributes to be defined as objects. An object can then be said to be composed from its state attribute values. This simple notion of composition is complicated when state attributes are defined as references to shared objects. Sharing is an efficiency matter which is not necessary for correct implementation of the object oriented designs.

- **Subclassing**

All object oriented languages offer a subclassing mechanism. This mechanism is essential for polymorphic properties to be offered in a controlled manner. Unfortunately, object oriented programming languages provide subclassing in the form of inheritance, which performs two distinct roles: it defines the class relationships in the system and defines how these relationships can be used to implement the efficient binding of 'shared code' to a service request. In this thesis, the subclassing relationships are essential to provide inclusion polymorphism, whilst the code sharing aspects are secondary to this main issue.

---

<sup>5</sup>This notion of attribute is different from our well defined notion of attribute (as part of an object's interface), but more of this difference later.

It is also essential that the subclassing is supported by a ‘multiple inheritance’ mechanism, since we require that a class can be defined as a subclass of different superclasses which themselves are not related by a subclassing relationship. (Unfortunately, the way in which object oriented programming languages cope with conflicts in multiply inherited attributes is not consistent.)

- **Substitution Polymorphism**

An object which is typed to be a member of a class, C say, must be acceptable as a member of any of the superclasses of C (this is the well accepted notion of substitutability). This type of requirement can be met by any untyped (ad-hoc polymorphic) language. However, this option is ruled out by the next essential characteristic: strong typing.

- **Strong Typing**

When each variable in a system is known merely to be an object, of some unspecified sort, this is known as weak typing. Contrastingly, in strongly typed object oriented languages, every variable is precisely defined as belonging to a particular class. Strong typing is essential, in our opinion, because it provides facility for actively supporting the implementation of correct code. Ideally, type correctness in the implementation language is guaranteed by type correctness in the LOTOS design. However, in practice, implementations have typing aspects which are not directly checked by earlier development stages.

### 6.2.2.2 Important Characteristics

The characteristics which we consider important, but not essential are:

- **(Incremental) Compilation**

It is important that the implementation code can be compiled into machine code. This requirement is purely an efficiency and portability concern. Incremental compilation is an additional advantage because it leads to the generation of autonomous re-usable implementation components.

- **Genericity**

Genericity is not a subclassing mechanism, but it is a powerful technique for defining parameterised behaviour. Genericity improves understandibility (by highlighting common structures) and encourages re-use.

- **Comprehensive Class Libraries**

Most object oriented programming languages include a library of standard classes for general purpose data structures, file handling, user-interfacing, graphics, mathematics, etc . . . . Without these class libraries, object oriented programming is very difficult.

### 6.2.2.3 Beneficial Characteristics

It is beneficial for the following characteristics to be offered by the chosen implementation language, but not essential at this early research stage.

- **Assertions**

Assertions can improve the mapping between requirements and implementation. Provided that the object oriented design assertions (defined as boolean expressions) can be expressed directly in the implementation language, there can be automatic checks, made during execution, that the requirements are fulfilled. Thus assertions (of some sort) in the implementation language can improve the testing process. A second consideration is that assertions improve the understandability of the code. Consequently, it is advised that assertions are placed in the code as comments, even when no mechanism exists for making the checks during execution.

- **Garbage Collection**

Dynamic object oriented systems require the production and destruction of component objects during execution. Garbage collection is an important memory management facility which frees unreachable object space for future use. Some object oriented programming languages do not provide automatic garbage collection, but expect programmers to explicitly deallocate memory when an object is no longer needed.

- **Wide Acceptance (in industry)**

It is important that we target our designs towards implementation languages which have a wide acceptance (industrial as well as academic). Widely used languages offer continual support via published work and second-hand user experience.

- **Packaging**

A class is not an ideal fundamental building block for re-use. In many cases it is beneficial to be able to re-use groups of related classes (a package). Packages can help to control visibility between classes. Object oriented programming languages often require unique class identifiers. This is counter-productive to the independent production of compatible re-usable classes. Packaging can provide a means of defining name-space domains to avoid this problem.

- **Concurrent Constructs**

Concurrent constructs have the potential to improve efficiency, increase resource utilisation and more naturally model the real world requirements of highly parallel systems. Concurrent constructs free designers from having to target the designs towards the constraining non-concurrent semantics which dominate programming languages at the moment. Concurrency also improves the extendibility of the system. Section 6.5 examines the issue of concurrency in more detail.

- **Tool Support**

Software development tools (for example, debuggers, browsers, interpreters and syntax directed editors) have the potential to improve productivity. Also, they can improve the chances of the code meeting customer requirements. Tool support is particularly important in an object oriented implementation environment [57].

- **Persistency Support**

A permanent data store is required by a large number of software systems. A persistency mechanism can simplify the implementation of a data store, and consequently make the code easier to understand.

- **Purity**

Object oriented programming languages are categorised as being pure or hybrid. Pure languages are those which do not provide language constructs whose roles are outside the object oriented paradigm. Hybrid languages, being extensions of non-object oriented languages, provide language constructs whose roles are not necessarily object oriented. The problem with many hybrid languages is that object oriented principles are not enforced. Further, the non-object oriented constructs can be used to violate the object oriented requirements. Purity also makes the implementation code much more consistent. In general, consistency implies coherency. (Smalltalk, by enforcing the consistent notion that *everything* is an object, is an excellent counter example to this claim.)

#### 6.2.2.4 A Note On The Importance Of Semantics

A separate problem occurs when characterising programming languages if their semantics for particular characteristics do not match the formal semantics in the formal designs. Eiffel and C++ provide two interesting examples of this:

- Eiffel offers a subclassing mechanism (inheritance), but this does not fulfil the contravariance requirement in the formal object oriented design model. In this case, Eiffel provides a subclassing mechanism, but does not fully support the subclassing requirements.
- C++ claims to offer polymorphism when what it actually offers is the dynamic binding of code to message requests. It does not offer replacement polymorphism.

These types of subtle semantic differences plague the process of translation between languages with different semantics, particularly when the target implementation language semantics are not formally defined.

### 6.2.3 A Review of OOP Languages

It is not possible to review all available object oriented programming languages. Five of the most popular languages, namely Simula, Smalltalk, C++, Eiffel and CLOS, are considered in sections 6.2.3.1 and 6.2.3.2, below.

#### 6.2.3.1 Overview of Language History

This section gives an introduction to each of the five languages by giving a brief review of their histories.

- **Simula** was designed in 1967 as an extension to Algol 60 [91]. It is a general purpose language which, although often ignored by object oriented programmers, is still widely used. Simulation is just one application of Simula.
- **C++**, an extension of C, was designed by Stroustrup in 1984 [106]. It is widely distributed in many forms (by commercial vendors and as public domain software). It is likely to be

the dominant object oriented programming language of the 1990s. The main weaknesses of C++ are: the lack of substitution polymorphism, its hybrid nature and its lack of support for organising libraries (different library classes often turn out to be incompatible). As with C, the syntax is awkward and difficult to parse. A good reference to the latest versions of the language is [2]. C++ is still evolving but there are standardisation efforts. Unfortunately, at the present, the semantics of C++ are such that formality is out of the question.

- **Smalltalk-80** was the first popular object oriented language, developed at Xerox Parc by Kay, Goldberg and Ingalls [58, 57]. It is best known for its contribution to the development of graphical user interfaces, and the manner in which it provides a programming environment (set of complementary tools) rather than just a programming language. Its purity is taken to extremes: all things are objects (even classes). This consistency, paradoxically, can be quite confusing to beginners and experts alike. It is an interpreted language which does not perform any strong type checking. It is a good language for learning about object oriented programming, but it is not suitable for large scale software development.
- **Eiffel** was developed by Bertrand Meyer in 1988 [84], in response to the need for a strongly typed but dynamically bound object oriented programming language. It has many innovative features and appears to provide all that one would require in an object oriented language, but it has its problems (particularly in its implementation). Eiffel is examined in more detail in section 6.4, as the language chosen for implementing object oriented LOTOS designs.
- **CLOS** (Common Lisp Object System) is an extension of common Lisp (see [73, 42], for example). It was developed to include the best features of a wide range of Lisp-based object oriented languages (e.g. Flavours [88] and CommonLoops [7]). Although it is a hybrid language, the object oriented language constructs are so well integrated with the Lisp features that it can be treated as if it was pure. CLOS adheres to the Lisp philosophy of flexibility: it is weakly typed and encapsulation is not enforced.

### 6.2.3.2 Comparing Characteristics

The table in figure 6.2 identifies the three types of characteristic (essential, important and beneficial) each of the five languages support and, in appropriate cases, the degree to which they are supported.

### 6.2.4 Choosing Eiffel

Examination of the table in figure 6.2 clarifies the reasoning behind choosing Eiffel for implementation of the formal object oriented LOTOS designs: it is the only language, under consideration, which fulfils all the essential requirements<sup>6</sup>. Eiffel is not ideally suited to our needs (see section 6.4), but is the *best* option available within the timescale of the thesis. Currently, work is being done towards automating the generation of Eiffel code from object oriented LOTOS designs. This thesis reports only on the manual production of code.

---

<sup>6</sup>C++ was originally used in the case study, together with Eiffel, but there were great problems with its lack of substitution polymorphism and its informal, yet very complex, definition.

language characteristic	Simula	C++	Smalltalk-80	Eiffel	CLOS
Classification	Direct	Direct	Direct	Direct	Direct
Encapsulation	Direct	Direct	Unsupported	Direct	Unsupported
Composition	Direct	Direct	Direct	Direct	Direct
Subclassing	Single	Multiple	Single	Multiple	Multiple
Polymorphism	Direct	Unsupported	Direct (ad hoc)	Direct	Direct (ad hoc)
Strong Typing	Direct	Direct	Unsupported	Direct	Unsupported
Compilation	Yes	Yes	No	Yes (incremental)	Yes
Genericity	Unsupported	Direct	Not Applicable	Direct	Not Applicable
Library Support	Minimal	Excellent	Excellent	Excellent	Minimal
Assertions	Unsupported	Supported	Unsupported	Direct	Supported
Garbage Collection	Yes	No	Yes	Yes	Yes
Industrial Acceptance	Minimal	Yes	Little	Little	Minimal
Packaging	No	Limited	No	No	Yes
Concurrency	Direct	Supported	Supported	Supported	Unsupported
Tool Support	Average	Good	Good	Good	Average
Persistency	Unsupported	Supported	Supported	Direct	Unsupported
Purity	Hybrid	Hybrid	Pure	Pure	Hybrid(used purely)

Figure 6.2: Characterising Object Oriented Programming Languages

## 6.3 Translating Design To Implementation: Mapping Semantics

### 6.3.1 Implementation Languages: The Importance of Semantics

Good programmers understand the semantics of their chosen implementation language. Bad programmers suffer from a lack of semantic understanding: they must continually check their understanding of the language.

#### 6.3.1.1 The (Reluctant) Acceptance Of Informality

The high-level object oriented designs are constructive in nature and, as such, have potential for direct compilation. A direct compilation approach is not advocated in this work because it is necessary for programmers to be able to manipulate and interact with the implementation code, and the object oriented LOTOS code (in its present form) is certainly not suitable for use by object oriented programmers. Consequently, this thesis advocates using a different language for implementation. There are difficulties which arise from this approach:

- The chosen implementation language Eiffel, like most object oriented programming languages, has no formal semantics. Consequently, there is no way to prove that the code is a valid implementation of the design (and therefore fulfils the initial requirements). Certainly, having

a formal object oriented design helps to test the code, but it cannot guarantee correctness.

- The informal semantics of Eiffel appear, at first glance, to correspond to the semantics of the object oriented LOTOS designs (using the procedural communication model). However, there are many differences between the design semantics and implementation language semantics (these are covered in more detail in section 6.4). The naive view is that the design and implementation languages share a common semantics (i.e. object orientedness) and this can lead to many problems. Implementations may appear to fulfil their requirements (as specified in the design), but without formal semantics these appearances can deceive.

These problems arise from informality in the programming languages. Programmers can, in principle, do what they want with the design provided they can verify their executable model against the requirements. In practice, this is of course impossible. A formal design phase is needed in software development to ensure the requirements are correctly stated in the design. Formally defined programming languages are necessary to guarantee the correct implementation of the design.

#### **6.3.1.2 Object Oriented LOTOS: A Formal Executable Model For The Future?**

An alternative to mapping the LOTOS designs to an existing object oriented programming language is to create a new formally specified programming language (based on the object oriented design semantics). Such a language could be a simple syntactic sugaring of the LOTOS object oriented design style of specification. Using a formal implementation language retains formality in the step from design to implementation. However, it was not the approach taken in this thesis:

- The thesis argues that formal object oriented development of software systems using LOTOS is possible. It is easier to show this by mapping the designs towards well accepted object oriented programming environments than by producing such an environment from scratch.
- It is hoped to transfer this work to industry. Industrial acceptance is difficult to achieve without a well accepted base: in this case, a mature programming language and environment provide a foundation upon which industrial interest can be developed.
- By carrying out the implementations in Eiffel, it is possible to show that our approach to software development is practical, whilst also emphasising the problems which can arise when the implementation language has no formal semantics.

There is a need for a formally specified object oriented programming language. A natural extension to this thesis is the development of a programming language based on the object oriented semantics of the formal designs.

#### **6.3.1.3 The Remote Procedure Call Communication Model: An Easy Target**

Wegner defines four fundamental types of object: functional, server, autonomous and slot-based, according to their external and internal communication models [124]. Eiffel objects are of the server type: the objects are active only when a message is received that triggers the object's internal

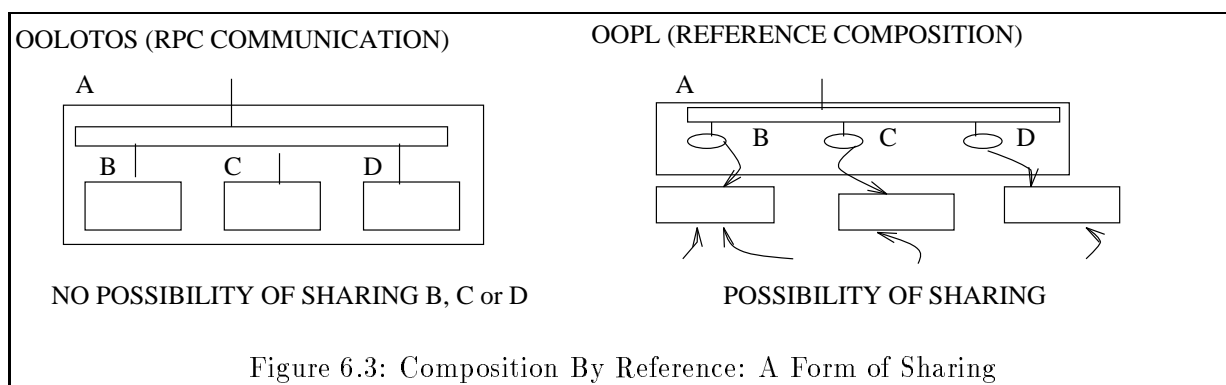
operations. These internal operations are themselves sequences of service requests to state attribute objects of the server. Consequently, there is a single thread of control in an Eiffel implementation. Control goes from client to server and returns to the client after the server updates its internal state and/or returns some result.

### 6.3.2 Peculiarities of LOTOS Designs

There are many aspects of the LOTOS designs which are peculiar to the approach advocated in this thesis. These must be kept firmly in mind when deriving object oriented code.

#### 6.3.2.1 Sharing

In LOTOS designs whose communication models are procedural, there is no notion of shared objects. For example, when an object **A** is composed from components **B**, **C** and **D** then there is no access to **B**, **C** and **D** except through **A**. In effect, all external events of **B**, **C** and **D** must synchronise with the control process of **A**. This is illustrated in the left hand side of figure 6.3. Contrastingly, in object oriented programming languages, it is common to be able to define components as references to objects<sup>7</sup>. This is illustrated by the right hand side of figure 6.3.



Consider a LOTOS design of a database enquiry system. There are two components: the database and the interface (which interprets user interactions). A new multi-user system is required to allow parallel access to three users, for example. This type of behaviour is most naturally implemented using sharing, even though it cannot be specified in that way using the formal procedural communication model. Rather than having all access to the database to be through the system control, sharing permits each individual interface to have a reference to the database (and thus have direct access to information). The shared database system can be said to be composed from a number of single-user database components. The underlying components in the design and implementation are the same: the three interfaces and one database. However, the way in which they communicate and interact is very different. This is illustrated in figure 6.4.

Sharing is a powerful implementation mechanism, but it breaks the principle of encapsulation. The state of each of the components of the shared database implementation can be accessed without

<sup>7</sup>In Eiffel, this is the only way of defining object components.

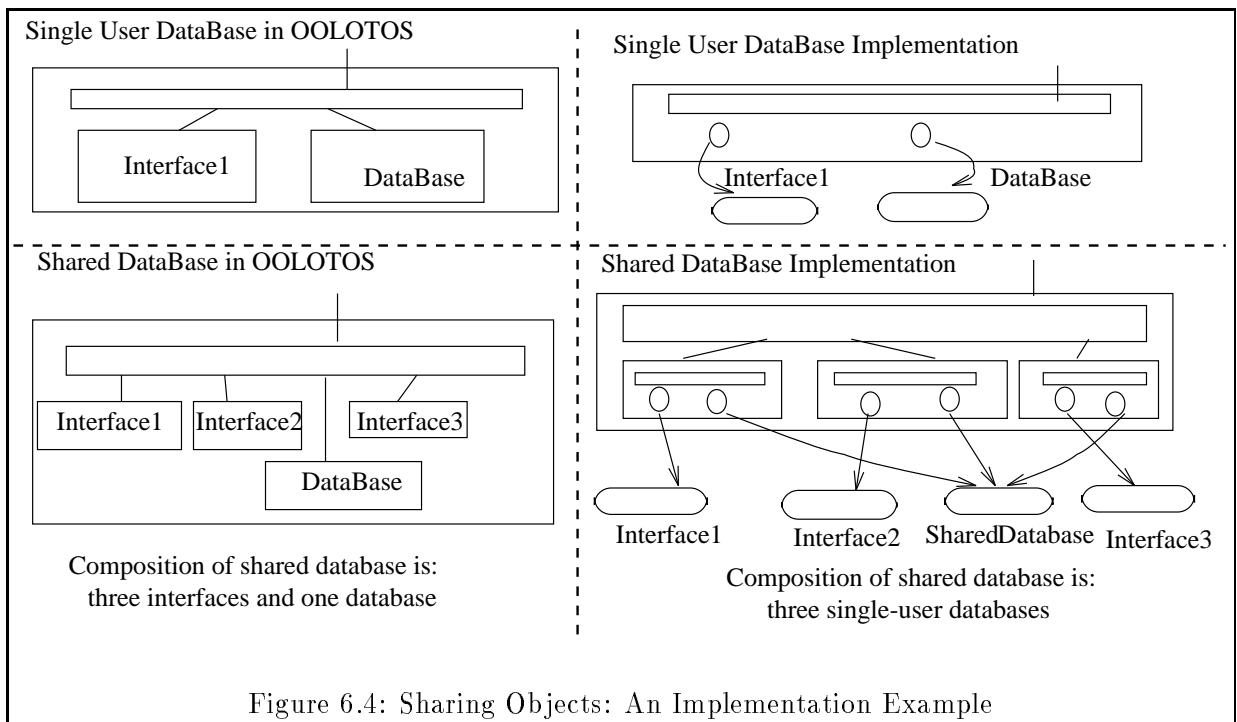


Figure 6.4: Sharing Objects: An Implementation Example

using the component's interface. Alternatively, one could argue that the database is not actually a component of any of the components but is a component of the whole multi-user system (a global variable). Sharing is examined in more detail in section 6.4, when the Eiffel reference semantics are considered.

### 6.3.2.2 Polymorphism: Parameter Replacement

The OO ACT ONE requirements model states that all parameters of an operation can be actualised by an instance of the specified class or by an instance of a subclass of the specified class. This property of object oriented systems is known as polymorphic replacement. The ACT ONE model specifies polymorphic replacement using coercion and operation overloading. The transfer of ACT ONE structure to the formal process algebra is complicated by the polymorphic requirements<sup>8</sup>.

Polymorphic replacement must be considered during implementation. In some languages, this type of polymorphism will be provided for automatically. However, in languages whose semantics do not provide this polymorphic property, it is necessary to code it explicitly where necessary.

### 6.3.2.3 Implementation Freedom

In the analysis model, a service request is defined as the evaluation of a **state label expression**. When this is translated to ACT ONE, a service request is defined as the evaluation (simplification using standard re-write rules) of an ACT ONE expression. Now, when a system services a request by

<sup>8</sup>The main weakness of using LOTOS for our object oriented design language is that it does not directly support replacement polymorphism.

using its components, these expressions will be constructed from subexpressions which correspond to internal service requests. Often, the order in which these subexpressions is evaluated is arbitrary. This is specified using the interleaving operator ( $\parallel$ ). In a procedural implementation, the implementer must resolve this freedom of implementation feature by removing the internal parallelism. This does not change the external behaviour.

#### 6.3.2.4 Exceptions

Exceptions, in the LOTOS designs, are handled in a very distinctive manner: exceptions are returned (as **unspecified** results) to the service requesters (clients) rather than resulting in a run-time error. For example, just because a stack is empty should not prevent it from servicing pop requests, otherwise static analysis cannot, in general, guarantee the absence of a run-time error due to an empty stack receiving a pop request. In the stack case, the designers can either chose to explicitly handle the exception or they can leave the programmers to cope with it. Exceptions are not necessarily error cases, they represent some abstract behaviour which is to be made concrete at less abstract stages of development. Implementers must handle all exceptions in a consistent and coherent fashion. Having exceptions in the design is useful if the target implementation language provides a mechanism for handling them.

## 6.4 Producing Eiffel from Procedural Object Oriented LOTOS Designs

### 6.4.1 Setting Reasonable Bounds

Implementing object oriented requirements using Eiffel is a large area of research in its own right. Further, the production of executable code for any given formal specification is a non-trivial task. It is not possible within the implementation part of this thesis to examine formal code generation in any detail. Rather, we set reasonable bounds for the implementation.

#### 6.4.1.1 Restricting Designs to The RPC Communication Model

Rather than attempting to show how Eiffel can be used to implement any given LOTOS design, we restrict ourselves to those written using the procedural communication model.

#### 6.4.1.2 Restrictions on the Eiffel Syntax

Eiffel is a large, complex language with many mechanisms, not all of which are a direct consequence of an object oriented philosophy. Rather than attempting to examine the complete language, only those aspects which are directly relevant to the implementation of formal designs are considered. This simplifies the process of code generation, but means that the code produced in this way may not be the most efficient. It is beyond the scope of this thesis to consider all the ways programmers can tune their code whilst retaining correctness, particularly with respect to the choice of certain language

constructs over others. We are primarily concerned with producing code which fulfils its functional requirements.

#### 6.4.1.3 Emphasis On Semantics

This section does not attempt to analyse the suitability of Eiffel as an object oriented programming language. Such an analysis necessitates:

- A study of Meyer's object oriented philosophy.
- A critique of the Eiffel environment (the language implementation and tools) with respect to their practical application in large scale software development.
- The undertaking of a variety of case studies using Eiffel.

Meyer, of course, gives his opinions on these aspects. A more objective view is given in [121].

#### 6.4.1.4 The Language Version

Eiffel is continually being updated and errors corrected. Meyer has listened to much of the criticism of the language and attempted to make improvements. Unfortunately, it is not possible to always have the most up-to-date version of the language. Further, it is often best to stick to using an older version, rather than continually changing ones understanding of the semantics. Many aspects of the language have remained constant, whilst other important features of the language are very unstable. Analysis of these features is restricted to an early version of the language, namely version 2.3. The reason for this being that the coding in the case study (see chapter 7) was carried out over a year before this research was written up. The newest version of Eiffel is defined by Meyer in [86].

### 6.4.2 Coding Design Requirements in Eiffel: An Overview

The production of Eiffel code is considered in three main sections:

- Modelling object based requirements.
- Modelling object oriented requirements.
- Utilising assertions and exceptions.

Then, some other interesting aspects of Eiffel are considered. This work is usefully preceded by a comparison between the semantics of Eiffel and the semantics of the object oriented LOTOS designs.

### 6.4.3 Reference Semantics vs Value Semantics

The object oriented LOTOS and Eiffel have fundamentally quite different semantics, even though they both have an object oriented flavour. The mapping between the two languages appears quite straightforward until the semantics are studied in more detail. Call-by-reference semantics, as is prominent in Eiffel, is appropriate for the specification of an executable language in which *how* not *what* is a prime concern. Further, call-by-reference gives much more control to the programmer with

respect to efficient allocation and use of resources. The value semantics, as is evident in the formal designs, is appropriate for the specification of behaviour at a higher level of abstraction (and whose efficient execution is not a prime concern). Before addressing the problems of relating the two different semantics, it is useful to examine the relative merits of each.

#### 6.4.3.1 Advantages of Reference Semantics

The advantages of reference semantics arise from the extra control given to a programmer with respect to the way in which memory is utilised. By using references, programmers can explicitly access and manipulate state rather than state variables. This is a powerful facility which is often abused. Eiffel reference semantics supports two very powerful programming techniques:

- **Sharing**

Sharing is necessary when state attributes of different objects must refer to the same object: as opposed to distinct but identical objects. Sharing leads to an economy of space, efficient memory access and update, and semantic integrity (if something in the shared object changes, then this change is simultaneously reflected in all the clients of the shared object). To implement sharing, state attributes are declared as references to other objects. In Eiffel, all state attributes (other than those of simple types) are implemented as references. The global scope of references in Eiffel means that any object can be referenced by any other object. A limited form of sharing control is provided by the Eiffel constant references and the **once** construct. This allows objects to be shared amongst instances of a particular class (and no other).

- **Linked data structures**

All programmers are familiar with the notion of linked data structures: stacks, lists, trees, etc . . . . Linked data structures are most useful when there is recursion, or even self reference. They provide the most efficient means of constructing large data stores, with high degrees of control over how the structure is traversed. Linked data structures offer a natural way of implementing recursively defined class structures.

#### 6.4.3.2 Disadvantages of Reference Semantics

Many complications arise when using references:

- **Creation and Initialisation**

Because the state attributes of an object can themselves be references to other objects, object creation and initialisation must be done in two steps: a declaration (for example, **x:X** declares **x** to be a reference to class **X** and sets the state of **x** to be **void**) and an association (for example, **x.Create** creates an object of class **X** and associates it with reference **x**). There is a confusing duality between the references and the instances.

- **Memory Management**

Object instances may, at run time, be unreferenced. It is necessary for this state to be made available for re-use (garbage collection). Eiffel provides an automatic garbage collection facility

(as opposed to requiring the programmer to handle it), but it does make the code produced much less efficient.

- **Dynamic Aliasing**

It is dangerous to have one object which is accessed through two different references. In particular, when references are passed as operation arguments (external attribute parameters) it is not possible to guarantee that the execution of the routine does not change the state of objects other than the one currently servicing the request.

- **Testing for equality and copying**

With references there are two ways of defining equality: by reference, or by ‘state’. Testing for equality of ‘state’ can be shallow (where all fields are tested for equality of reference) or deep (where all fields are tested for equality of ‘state’). In some complex, recursive data structures it is necessary to define even more complicated equalities. The same complexities arise when one considers defining assignment and cloning.

### 6.4.3.3 References and Values: a Logical Equivalence

It is very simple to implement call-by-value semantics in a call-by-reference language: all calls to a reference are simply replaced by making a deep copy of the object being referenced and passing a reference to the new copied object. In this way, the absence of sharing is guaranteed.

It is also simple to implement call-by-reference semantics using a call-by-value language: every object is uniquely identified and kept (together with its identification) in some global data store. The object identifiers can then be used as state attributes of other objects. Access through identifier can be provided by a global system function which is visible to all objects.

The advantage of a reference semantics is that the global state (and global means of allocation and access) are provided by the language rather than needing explicit control by the programmer.

Reference semantics are very powerful, but do make the production of correct code much more difficult. When implementing the formal designs we do not advocate the ‘do everything by value’ approach. However, we do not wish every object to be made available for sharing. Consequently, we advocate the use of sharing only in special cases and code the Eiffel so that, by default, operation arguments are passed as references to copies rather than sent as references to the actual parameter. In special cases, sharing can be contained within predefined classes: trees, lists, rings etc .... For example, a linked list structure can be used to implement the following `List` class behaviour.

```

Class List USING listelm OPNS
LITERALS: empty
STRUCTURES: ListStr <List, listelm>
...endclass (* List *)

```

An Eiffel implementation has a structure as defined below.

The object, `AList = List(List(List(empty,3),2),1)` can be created in Eiffel as follows:

```

AList, BList, CList, DList: List
CList.ListCreate(DList, 3); -- By default DList is empty

```

```

class List ...
feature
  Next: List;
  Elm: ListElm; ...
end -- class List

```

```

BList.ListCreate(CList, 2);
AList.ListCreate(BList, 1);

```

#### 6.4.4 Coding Object Based Requirements

##### 6.4.4.1 A Class Instance Is A System

Meyer states that:

‘The absence of a notion of main program and of any structuring mechanism at a higher level than the class is an important element of the Eiffel software design philosophy.’

Each class is an executable entity in its own right. The process of creating a system from a class is called assembly. A system is characterised by a *root* class. System execution is done in two steps: *root* declaration and execution of the *root* class *Create* routine.

This suggests that every class can somehow be instantiated and executed. However, in most cases, the creation routine of a class just instantiates the state attributes. Such a creation routine shall be known as a **base creation**. The execution of a **base creation** will not produce a system that performs any useful purpose (other than its very existence). Classes which are intended to be systemised (i.e. turned into systems) are more commonly defined to have a creation routine which acts as a type of main program.

It is not desirable for all classes to have create routines that act as programs in their own right. It is desirable, however, that system classes can be generated from any given class corresponding to a component in the formal design. Object oriented Eiffel implementations generally have a complex root creation routine which provides the user-interface to the system. The coding of such an interface is important for the system, but it is not important for subcomponents of the system. Rather than having programmers define system containers for every class, it is useful to provide a default mechanism to produce a primitive system with minimal human-computer-interface (HCI) features. Such a mechanism is very easy to develop.

##### 6.4.4.2 Defining Class Members

Class members are defined by the different states which objects of a class can attain. In other words, objects are references to particular class members, and every object of a class is uniquely identified with one class member. In Eiffel, every class has a fixed set of state attributes. This is inflexible:

- Literal values cannot be directly defined.
- Classes with 2, or more, structures cannot be directly represented.

Literal values are the object oriented equivalent of enumerated types. Eiffel does not support enumerated types: they must be declared as literal constants (normally integers). For example, the class `t-light`, with three literals `red`, `amber` and `green`, is implemented in Eiffel below.

```
class t-light ...
feature
red: INTEGER is 0;
amber: INTEGER is 1;
green: INTEGER is 2;
t-light-state: INTEGER ...
```

Multiple structures can be implemented by defining a structure enumeration to identify the structure (or literal) currently being used as the state constructor of the object. This requires the parameters of every structure operation to be included in the state attribute set (of the Eiffel code), with only a subset of the parameters having meaning at any particular moment in an object's life-time. For example, consider the partial behaviour of the `Student` class, defined below.

```
CLASS Student USING ...OPNS
LITERALS: unregistered
STRUCTURES: single<subject>, joint<subject, subject> ...
```

This is implemented in the Eiffel `Student` class below.

```
Class Student ...
feature
unregistered: INTEGER is 0;
single: INTEGER is 1;
joint: INTEGER is 2;
student-state: INTEGER;
single-subject1, joint-subject1, joint-subject2: subject; ...
```

The state of an object is determined by the `student-state` attribute, together with the relevant parameter attributes. The naming convention for the structure parameter state attributes makes it simple to identify the role of each state attribute: the structure name is followed by the parameter class name and index. (Note that there is a redundancy when different structures have common parameters, but it is better to keep the redundancy as it improves the mapping between specification and implementation.)

#### 6.4.4.3 Defining Class Interfaces: Exporting Eiffel Features

Eiffel groups state attributes and operations (methods) together as **features**. By default, all **features** are private (not part of the external interface). Public features must be listed in the **export** clause at the beginning of the class definition. Meyer does not distinguish between state attributes and operations because, as he correctly argues, it does not matter whether a feature is stored as a state or provided as an operation. This is certainly true, but we believe that distinguishing between state and

operation is important since the state structure corresponds to our notion of composition. The state attributes can still be offered as features by defining particular operations for this purpose. This is the approach taken in our Eiffel implementations: none of the state attributes are exported (directly) and there is a one-to-one correspondence between the exported Eiffel features and the external attributes of the design classes. Eiffel procedures are used to implement transformer attributes, whilst Eiffel functions are used to implement duals and accessors: duals are implemented as functions with side-effects.

A simple stack class illustrates the way in which an object oriented class specification is implemented in Eiffel. The OO ACT ONE code for the `Stack` class is given below.

```

Class Stack USING Int OPNS
LITERALS: empty
STRUCTURES: St<Stack, Int >
TRANSFORMERS: push<Int>
DUALS: pop - > Int
EQNS
Stack1.push(Int1) = St(Stack1, Int1)
empty.pop = empty AND 0; (* ignore exceptions, for the moment *)
St(Stack1, Int1).pop := Stack1 AND Int1
ENDCLASS (* Stack *)

```

This behaviour is implemented in Eiffel below.

This simple implementation of stack behaviour illustrates some interesting points:

- The `INTEGER` base type is used to provide the behaviour of class `Int`. Base types are reconsidered in section 6.4.7.
- The `Create` operation is the default means of initialising the state of an object. Eiffel does not permit creation routines to be overloaded and so we cannot define one creation routine for each literal and structure of a class. In some specifications, like the one above, it is useful to be able to create an object as an instance of a literal or structure. In such cases, we define operations *LiteralNameCreate* and *StructureNameCreate(...)* in an appropriate fashion.
- The code for the implementation of each of the external attributes (`push` and `pop`) is more complex than the specification code because extra care is needed when coding with references in Eiffel.

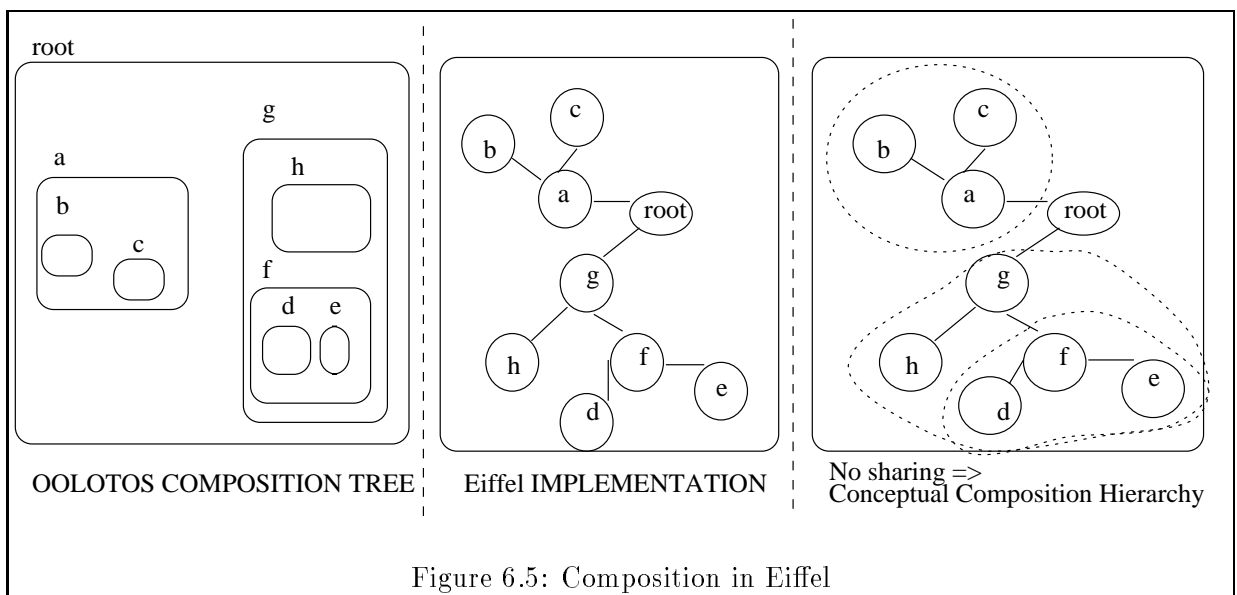
#### 6.4.4.4 Composition

Every structured object in the formal design is represented as a process in which the structure is either specified in the ADT part, or specified as a set of component processes. The structured objects in the formal designs can be said to contain their components. In Eiffel the components (state attributes) are defined as object references. Therefore, to implement containment (encapsulation) we implement every state attribute as a unique reference which is never passed out of the containing object. In this way, complex Eiffel systems (without sharing) are given a conceptual composition hierarchy based on encapsulation. This is illustrated in figure 6.5.

```

class Stack export push, pop
feature
  empty: INTEGER is 0;
  St: INTEGER is 1;
  Stack-State: INTEGER;
  St-Stack1: Stack;
  St-Int1: INTEGER;
  emptyCreate is do Stack-State := empty end
  StCreate(Stack1: Stack, Int1: INTEGER) is do
    Stack-State := St;
    St-Stack1 := Stack1;
    St-Int1 := Int1 end
  pop is local tempstack :Stack do
    if Stack-State = empty then Result:= 0 else
      tempstack.Create;
      tempstack:= Current;
      St-Stack1:= tempstack.St-Stack1.St-Stack1;
      St-Int1:= tempstack.St-Stack1.St-Int1;
      Stack-State:= tempstack.St-Stack1.Stack-State;
      Result:= tempstack.St-Int1
    end; -- pop
  push (int1: INTEGER) is local tempstack: Stack do
    tempstack.Create;
    tempstack := Current;
    Stack-State:= St; St-Stack1:= tempstack; St-Int1:= int1;
  end; -- push
end -- class Stack

```



By convention, unshared component attributes are commented as: `-- components`. The pro-

programmer must then ensure that such references are never passed outside the containing class. In the special cases where sharing is required, or a linked data structure is being used, then these shared attributes must be commented with an explicit statement of why and how the reference is being shared (this is good programming practice).

#### 6.4.4.5 Implementing State Changes

The state changes of a structured process are implemented as follows:

- **Restructuring**

This is implemented in Eiffel by changing the enumerated structure value and updating the other state attributes appropriately.

- **Pure Updates**

These occur when the structure representation remains the same and the state of the system changes only as a result of service requests to component objects changing the components' state. In Eiffel, this means that the state attribute references do not need to be directly manipulated.

- **Impure Updates**

These occur when the structure remains the same, but at least one of the components changes because of direct manipulation. In Eiffel this can only be implemented by an explicit change to one or more of the state attribute references.

During design, emphasis is placed on purity: such classes can be very simply implemented in a reference semantics because no direct manipulation of the referenced component attributes is necessary. Impure state changes are prone to errors because the state attribute references need re-allocation. In some cases, the extra work required of programmers does not justify the flexibility of allowing impure changes to state. Contrastingly, restructuring of state corresponds to special events in the lifetime of an object and therefore the additional work required on the programmers part is justified.

#### 6.4.4.6 Implementing Encapsulation

It seems strange, when encapsulation is fundamental to object oriented programming, that object oriented programming languages use reference semantics in such an uncontrolled fashion. The Eiffel implementations of the formal designs use shared references only in very particular instances. Sharing cannot be discarded, but should be used only with due care and attention. Implementations which used shared references must always contain the sharing within well understood encapsulated behaviours. When such behaviours are re-used there is no evidence of the sharing at the component interface.

#### 6.4.5 Coding Object Oriented Properties

Subclassing (extension and specialisation) in the formal object oriented designs is concerned with two aspects of behaviour:

- **Behaviour Compatibility:** An instance of a class always offers equivalent behaviour as the corresponding instance of the superclass.
- **Polymorphic Replacement:** A subclass is type compatible with all its superclasses, i.e. type checking involves checking the class of actual parameters against all the subclasses of the formal parameters (including, of course, the class of the formal parameter).

Inheritance, in object oriented programming languages, is primarily concerned with code re-use and dynamic binding. Both these are efficiency concerns: the faster production, compilation and execution of code. However, the ability to override inherited methods means that behaviour compatibility is no longer guaranteed.

#### 6.4.5.1 Multiple Inheritance

Both OO ACT ONE (and the LOTOS designs in which OO ACT ONE is used) and Eiffel offer multiple inheritance. However, multiple inheritance in Eiffel has many associated difficulties which are not present in the design language. These must be addressed when implementing the formal requirements:

- **Repeated Inheritance**

All multiple inheritance systems must be able to cope with repeated inheritance: when a class has a superclass which can be reached by more than one route up the class hierarchy. Eiffel resolves this problem by adhering to the following rule:

*In repeated inheritance, any feature from a common parent is considered shared if it has not been renamed along any of its inheritance paths. Any feature which is renamed at least once is considered replicated.*

Consequently, when implementing repeated inheritance properties it is necessary for the Eiffel programmer to rename some features.

- **Name Clashes**

Languages with multiple inheritance must deal with name clashes, i.e. when features inherited from different superclasses have the same name. In Eiffel, name clashes are forbidden. This places the onus on the programmer to cope with name clashes (using a construct for renaming inherited features) and weakens ones ability to re-use classes (by inheritance) without undue problems. This renaming approach is definitely not an ideal solution [121] and newer versions of Eiffel [86] allow explicit routing qualification to cope with name clashes.

#### 6.4.5.2 Implementing Extension and Specialisation

Implementing the extension and specialisation relationships in Eiffel appears to be straightforward:

- **Extension**

An extension class is defined by including the new function in the subclass, and perhaps re-defining other features to take advantage of the new function for improved efficiency.

- **Specialisation**

A specialisation is declared by defining a new invariant of the superclass. This invariant identifies the particular state partition of the superclass which makes up the new state of the subclass. The subclass state enumeration is useful in this respect.

This subclassing seems to be a perfect match until one considers the rules of contravariance and covariance. The formal design semantics state that:

- The class parameters of a structure operation of a subclass may be defined as subclasses of the corresponding parameters in the superclass.
- The result of a valued transition in a subclass may be defined as a subclass of corresponding result in the superclass.
- The parameters of a transition in a subclass may be defined as superclasses of corresponding parameters in the superclass.

Unfortunately, Eiffel does not support the last contravariance property. Cardelli [22] provides an in-depth study of the reasons for and against contravariance. Cardelli advises that subclassing must support contravariance otherwise the polymorphic replacement property is not guaranteed to maintain correctness. Unfortunately, contravariance is very difficult to implement cleanly in Eiffel: it is necessary to override the Eiffel typing system with ones own (as in an imperative implementation) and graft this onto the Eiffel class hierarchy (a non-trivial task). Consequently, we chose to ignore the need for contravariance and keep it in mind for future development.

#### 6.4.5.3 Typing Problems

The type checking of the formal designs should, in theory, guarantee type checking correctness of the resulting Eiffel (except the instances of contravariance in subclassing definitions). However, implementation often produces code which requires its own type checking. Eiffel type checking is very complex. The implementation of Eiffel, to date, cannot cope with the checks required by the informal semantics. This is design oversight. Cook [30] analyses the holes in Eiffel type checking and proposes removing some Eiffel flexibility as a solution. The thesis by Dinesh [47] analyses Eiffel type checking with respect to incremental development.

#### 6.4.6 Using Eiffel Assertions and Exceptions

Assertions define properties of some value(s) of program components. The Eiffel assertion language is not as powerful as full predicate calculus but it does provide a means of defining simple boolean expressions.

Assertions take three forms in Eiffel:

- **Preconditions**

The **require** construct places a precondition on a routine being executed. This is not important when implementing the formal designs because the class instances are always able to service all requests in their external interfaces.

- **PostConditions**

The **ensure** construct is used to guarantee some property after a request has been serviced. Postconditions can be used to record some of the abstract behaviour defined for the class in which the routine is found. For example, after a **push** on a **Stack**, a suitable postcondition is the boolean expression **Stack.notempty**.

- **Class Invariants**

Preconditions and postconditions describe only properties associated with individual routines. We require, when implementing formal requirements, a means of implementing class invariants: global properties of class instances. Eiffel provides a **class invariant** construct. Class invariants are useful in our implementations because they are checked on creation and at all stable times in the lifetime of an object. Further, they are passed down the inheritance hierarchy. This is important for the correct implementation of specialisation relationships.

Assertions are advantageous in our object oriented implementations because they:

- Improve the relationship between specification and implementation.
- Act as a documentation aid.
- Can be used to explicitly handle exceptions in a coherent fashion.

In principle, runtime checking of assertions should not be necessary. However, it is currently beyond the state-of-the-art in software development tools to perform a static analysis of such correctness. Consequently, it is necessary, whilst testing, to monitor for exceptions in the Eiffel code at run time. After testing, this monitoring can be turned off to improve performance.

Note that assertions and exceptions are useful for implementers both as a debugging mechanism and as a way of relating requirements with code. However, the Eiffel mechanisms do not constitute a formal approach to software development: the mechanisms are purely syntactic sugar. Eiffel does, however, encourage a methodological approach to exception handling, which can be used to implement the exceptions in the formal designs.

In conclusion, we note the limitations of the assertion language in Eiffel. For example, in the **Stack** behaviour it is not possible to assert that **pop(push(x, s)) = s**. Meyer acknowledges this problem by stating that this behaviour should be incorporated as a comment. When implementing the formal designs in Eiffel, we recommend that the complete code for each class is included as a comment (or comments) in the Eiffel code.

#### 6.4.7 Other Aspects

Other important issues when coding with Eiffel are:

- **Genericity**

Inheritance and composition are not powerful enough for general re-use. A mechanism for defining parameterised classes (generic behaviour) is required. Generic classes are simple to define using Eiffel. The mapping from generic ACT ONE types is straightforward.

- **Packaging**

The lack of a packaging facility is one of the major weaknesses of Eiffel. Much of the abuse of multiple inheritance results from the need to inherit shared features which would be better grouped in some sort of package construct.

- **Simple Types**

Eiffel simple types (`INTEGER`, `BOOLEAN`, `CHARACTER` and `REAL`) are called-by-value. They are useful in providing efficient implementations of well-understood behaviour. However, there is one problem with them: they cannot be placed in the class hierarchy. When it is necessary to provide these behaviours in the class hierarchy (for example, one may wish to define a class `EvenInt` as a specialisation subclass of `INTEGER`) then these simple types can be contained within classes which contain the simple types as their only state attributes. This results in a loss of performance, but improves the consistency of the code.

- **Persistence**

Eiffel supports persistence using classes `STORABLE` and `ENVIRONMENT`. They are very useful, but again inheritance is often abused to utilise these mechanisms. These classes are most useful in the definition of the root system class.

- **External Interfacing**

The `EXTERNAL` mechanism provides for the incorporation of non-Eiffel code into Eiffel programs. It is beyond the scope of this thesis to examine this mechanism.

## 6.5 A Question of Concurrency and Distribution

This section is primarily concerned with concurrency, and the potential for application of the formal object oriented development method towards a concurrent implementation. Distribution of implementation resources is also fundamentally a concurrency issue: distributed systems are constructed from concurrent processes<sup>9</sup>. Utilisation of distributed resources can be a performance issue, particularly when the problems being solved are highly concurrent in nature. Parallel code can be used to solve some problems much more efficiently than others but, in general, this is not the case. There is a much more common reason, other than efficiency, for requiring concurrent software: the external interface of the system being coded may be physically distributed. By considering concurrency in object oriented implementations, the problem of distribution is also being addressed, albeit indirectly.

The preceding sections of this chapter have addressed only sequential implementation, and chapter 5 shows how designs can be targetted towards such an implementation environment. There are three main reasons for concentrating on sequential implementation, at this stage of the work:

- Eiffel does not directly support concurrency<sup>10</sup>.

---

<sup>9</sup>The problem of implementation in a parallel architecture and implementation in a distributed architecture are logically the same, but of course there are additional problems in distributed systems: for example, the inter-process communication is slow and may be prone to errors.

<sup>10</sup>Meyer is currently working towards a *concurrent Eiffel*, but this extension is not available commercially.

- Concurrent semantics are, by their nature, much more complex than procedural (sequential) semantics.
- The current state of the art in concurrent object oriented programming languages are neither robust enough nor well enough supported for the implementation stage of a general software development method.

In general, the following principle should be followed:

*Don't make object oriented programming more difficult than it needs to be: if concurrency is not necessary then don't introduce it.*

However, if a concurrent implementation is required then this section argues that formal object oriented development using LOTOS can help to deal with it. In particular, the CPT-based design process has great potential for application towards concurrent software development.

Object oriented development adheres to the most elemental engineering principle: *make the solution look like the problem*. Implicit in the formal analysis models is the notion that components of a system are concurrent. Designers must decide whether to ignore or utilise this feature. Generally, designers chose to ignore the concurrency because they know that it is much more complex to deal with than sequentiality. Concurrent designs should be specified only when the target implementation environment is capable of coping with such requirements.

### 6.5.1 Concurrency and Objects: Opposing Views

The object oriented community is divided on the notion of concurrency. The two extremes to this division can be categorised as *optimistic* (or naive) and *pessimistic* (or conservative).

- **The optimistic view:**

Object oriented approaches, rather than placing control with some sort of master process, offer a means of handling concurrency which is quite different from traditional approaches. In the object oriented paradigm, objects can 'look after themselves' within a concurrent environment. The shifting of responsibility from centralised control to decentralised control is natural in an object oriented approach. Further, such distributed control can simplify the system.

- **The pessimistic view:**

Concurrency issues, contrary to initial expectations, are not orthogonal to object oriented concepts. The interference of concurrency and object oriented features makes it difficult, if not impossible, to combine them in a consistent and coherent fashion. In particular, the efficient implementation of concurrent object oriented semantics is in doubt.

The remainder of this section attempts to give a more balanced view of the future of concurrent object oriented languages. In particular, it examines the potential benefits of using formal object oriented development when heading towards a concurrent implementation language.

## 6.5.2 Concurrency: A Problem of Scale

### 6.5.2.1 The Problem

Objects may be considered as independent abstract machines that interact in response to service requests. To consider each object as an independent concurrent unit (process) results in an overdose of concurrency: hundreds, thousands or even millions of concurrent entities are required in even the simplest software systems. Implementation environments have not yet reached the stage where such high degrees of concurrency can be adequately dealt with (in hardware or software). Consequently, such a naive approach to modelling concurrency will result in poor performance (if any performance at all). Further, the complex structure of a program is difficult to analyse as a large set of interacting concurrent behaviours.

### 6.5.2.2 A Solution: Mixing Communication Models

There is a simple way of coping with the problem of scale, without having to reject concurrency as an implementation strategy. Instead of every object in the system being implemented as some sort of independent abstract machine, the designers must explicitly identify (sub)systems of concurrent objects. Then, only particular parts of the system (hopefully the ones whose concurrent implementation would be most beneficial) need be modelled concurrently.

The formal CPT driven object oriented design method provides the flexibility to deal with such a distribution of concurrency. Throughout the LOTOS designs there are, at the moment, three standard internal communication models which structured processes can adhere to: procedural, centralised concurrency and distributed concurrency (see 5.5.5). Object oriented designers can, using LOTOS, mix these communication models throughout the design. Using such a flexible design technique means that concurrent aspects of the requirements model can be explicitly mapped onto concurrent resources in the implementation.

## 6.5.3 Concurrency and Object Orientation: Resolving Conflicting Requirements

It has been said that concurrency has no respect for the spirit of the object oriented paradigm [95]. Papathomas, in his thesis, identifies five requirements for satisfactory integration of concurrent and object oriented features:

- Mutually exclusive protection of object's state.
- Request Scheduling Transparency.
- Internal Concurrency.
- Reply Scheduling Transparency.
- Compositionality and Incremental Modification.

He argues that a concurrent object oriented language cannot offer abstraction, encapsulation and subclassing if it does not fulfil these requirements. However, a concurrent object oriented semantics which fulfils these requirements may not be amenable to efficient execution. Further, such semantics

may not be suitable for use throughout the whole of the development process. We believe that the object oriented LOTOS designs can be used to model object oriented properties and concurrency in a consistent and coherent fashion.

#### 6.5.4 The Future: Formality in Concurrent Compilers?

A design technique in which different communication models are distributed throughout the design is inflexible with respect to matching resources to design components. This matching must be done at compile time and so the mapping of processes to processors is static. Although the area of real concurrent compilers, where the compiler actually maps components of the design onto real hardware resources (chips/processors), is still in its infancy, it is clear that static allocation of resources has major disadvantages:

- When a resource fails, the whole system is affected.
- When more resources become available (which may be the case in a truly open distributed system) it is not possible for the compiled code to take advantage of this.
- As more demand is placed on the set of shared resources, it is not possible to release resources for other users.

Correctness preserving design transformations may hold the key to flexible concurrent compilation:

- Given a static concurrent compiler, it is important that different design models are tested before a final design is chosen for implementation. Such testing can evaluate the performance of the system when the concurrency is distributed in different ways. Then, when the final design is implemented, it is more likely that the resources are used effectively.
- The future for concurrency, and object oriented programming, may be flexible compilation, where the mapping from system component to resource is dynamic. It may be possible to compile the design to produce a virtual machine which can be executed in a number of different forms. The different forms can be virtual designs, composed from virtual processes, whose behaviours are related by a set of correctness preserving transformations. The executable machine must be aware of resource allocation in the implementation environment and change virtual design form in response to an increase in supply (or demand) of implementation resources. Of course, there will be overheads in performing such virtual form changes, but it is possible to envisage a case when such overheads would be negligible compared with the increase in performance. Perhaps some form of process caching, working in the same way as state caching, will become standard in such systems.

The point being made is not that these notions of flexible compilation are new, but that the formal object oriented design framework provides a semantics upon which such compilers could be constructed. Formality is essential in proving that changes of design form (in the virtual machine) do not affect the external behaviour of the system. Correctness preserving transformations appear to be an ideal theoretical tool for reasoning about such models. This (hypothetical) work is beyond the scope of this

thesis, but it acts as a good motivator for using formal methods: such concurrent compilation would be impossible to control without underlying formality. It is difficult to predict the future, particularly with regard to concurrency and objects, but formality is sure to play a major role.