

Chapter 5

Formal Object Oriented Design (Using LOTOS)

This chapter is structured as follows:

- **Section 5.1: Introducing Design**

This section introduces design. It argues that design is difficult because it is a combination of artistic and scientific abilities. Design quality is introduced, and the important difference between functional and nonfunctional requirements is briefly explained. Finally, software design is introduced: a short historical background is given, together with the identification of problems unique to software design and statement of intent to reuse work in other design areas, whenever possible.

- **Section 5.2: Learning From other Design Areas**

This section compares software design and other types of design. Software design is shown to have the problem of coping with change. Design principles and techniques, common to all design areas, are identified: the importance of language, the role of structure, the advantage of re-use and the necessity of testing. Finally, this section identifies engineering as the area outside computing most closely related to software design.

- **Section 5.3: Object Oriented Software Design**

Section 5.3 examines object oriented software design. Initially it gives an overview of software design by listing a set of general software design criteria and principles. Then, an explicit definition of the roles of object oriented designers is proposed. Finally, testing, verification and correctness preserving transformations (CPTs) are introduced.

- **Section 5.4: Object Oriented Design with LOTOS**

This section considers object oriented design with LOTOS. We argue that LOTOS is a good candidate as a formal design language, even though it was not developed for this purpose. The importance of balancing the roles of the process algebra and ADT parts of LOTOS is stressed. Finally, the problems in defining an object oriented style of specification in LOTOS are considered.

- **Section 5.5: FOOA As Input To Formal Object Oriented Design**

Section 5.5 formulates the initial step from object oriented analysis to object oriented design. Four different translations for mapping from OO ACT ONE to full LOTOS are considered. An object oriented interpretation of the first LOTOS specifications, produced using two of these transformations, is then given. The initial step from analysis to design is shown to be concerned with making concrete the semantics of object communication and interaction. Two models of control flow are chosen for particular attention: a remote procedure call model and a parallel model.

- **Section 5.6: Correctness Preserving Transformations (CPTs): Formalising Design**

After introducing CPTs with respect to design in general, and design with LOTOS in particular, the fundamental concepts are reviewed: design trajectory, implementation relation, verification and CPT formulation. Internal and external properties are distinguished and this leads to a simple separation of CPTs into functional and nonfunctional categories. The importance of well defined non-standard semantic views of LOTOS (and their graphical representation) is re-iterated. Finally, the CPT design trajectory is introduced as forming the basis of an ideal object oriented design method, which this thesis goes a small step towards achieving.

- **Section 5.7: A Set of Object Oriented Design Decisions as CPTs**

This section formulates five types of transformation for application during the design stage of FOOD: static structure expansion (decomposition), compositional restructuring for re-use, restructuring for distributing control, removing explicit nondeterminism and removing parallelism. In each case, the correctness of the transformations is discussed.

5.1 Introducing Design

Design, in general, is viewed as an artistic or creative process which combines natural ability with experience. It is found in many spheres of human activity, but it is far from being well understood and often seems inaccessible to the layman. The principles and practices which are applied to the design of software have a strong affinity with more traditional engineering: there is a subtle blend of scientific criteria with intuitive decision making.

The question of why design is difficult needs to be addressed. Most complex systems seem to have been built for a particular purpose. The designers of such systems obviously have this purpose in mind throughout the whole design process. In a sense, this type of design can be said to be *targetted*.

5.1.1 Design: The Creative Process

Design is a creative process concerned with decision making¹. Designers look for solutions to problems. They search a solution space to arrive at a final design. The way in which the search is carried out may be methodical, but never deterministic. To design is to blend the old with the new: designers

¹Thus, to do something by design means to do it by choice.

must use their experience and previous work (the old) to find a solution to their problem (the new). The creative side of design can be categorised as mixing three different modes of work:

- 1) Creating new components which are variations on already existing components and combining these new components in well-accepted ways (or structures).
- 2) Finding new ways of using components or combining components.
- 3) Gaining insight into a problem and building a design (component) to utilise this insight.

Most designers work in the first mode. For example, car designers create new cars by designing some new parts, utilising existing parts, and combining them in well established ways. Fewer designers work in the second mode. In the construction industry, for example, buildings with original structure (or layout) can be created from standard components. Building designers are aware of the way in which standard components can be combined and tend to concentrate on structure rather than individual components. The third mode of design is the rarest — perhaps these types of designers are better termed inventors?

5.1.2 Purposeful Design

In purposeful design, the designers have some goal to aim for and this goal is evident throughout the design process. Designers are involved in each design step in an attempt to reach their goal. Central to purposeful design is the customer requirements. There are two extremes to the way in which designers can develop understanding of the requirements:

- Designers perform their own problem analysis to develop an initial requirements model.
- Designers accept a requirements specification in which the requirements are completely and consistently recorded.

In practice, design occurs somewhere between these two extremes. This thesis argues that design should not involve an analysis of the problem domain, although it does involve analysis of the requirements model. It is the role of designers to restructure the requirements model to best use the resources in the target implementation environment.

5.1.3 Design Quality and Criteria

A design provides, as an end product, one possible solution to a problem. The design describes the structure of the solution by defining:

- A set of components.
- The relationship between components.
- The method of construction, i.e. a means of realising the component relationships.

Given a design, it is necessary to be able to assess its *quality*. In other words, what is required is a set of criteria by which a design can be judged. The first and foremost test must be whether it fulfils

the requirements as specified by the analysts. Designs which do not fulfil their requirements are said to be *unacceptable*.

Requirements are traditionally divided into two groups: functional and non-functional (for a more complete analysis of the difference between these groups see [67]). Functional requirements are those specified in the requirements model. Non-functional requirements are usually concerned with costs, physical constraints, previous practices, political issues, etc A design which fulfils both type of requirements is said to be *acceptable*. Given two, or more, *acceptable* designs we must ask how a designer chooses between them. In such cases it is difficult to be objective. To say that one *acceptable* design is *better* than another is a subjective statement based on the (usually informal) criteria upon which judgement is made. In some less technical environments it would be called *taste*.

5.1.4 Introducing Software Design

5.1.4.1 A (Very) Brief History Of Design

In the beginning, ‘programmers’ analysed, designed and coded (they still do, in some instances). The 1970’s saw the beginning of structured programming [45, 127, 44], which identified the need for a method to software production (coding). Structured programming evolved into the widely used set of different, though fundamentally similar, structured design methods. By the 1980’s, structured design methods were widespread and their usage well documented [80, 94, 27, 36, 51, 41]. At this point structured designers designed and analysed². The appearance of object oriented analysis and design methods, in the late 1980’s and early 1990’s [31, 25, 26, 13, 101, 84], were a consequence of the acceptance of object oriented principles within the programming community, and the transfer of these principles to the earlier stages of software development.

5.1.4.2 Software Design: Too Difficult For Words?

Software designers are faced with a unique set of problems:

- The requirements they are given make up a set of the most complex systems ever created by man.
- Software requirements are dynamic.
- Software design tools and methods are accessible to anyone with little experience in software development. When *bad* designers apply such methods, it is often the methods which get blamed for the resulting chaos. Software design is a complex process. The tools and methods are important, but designers must understand the underlying complexity of what they are being asked to produce and the principles behind the methods they employ. This is not always the case. Complex software designs can be produced very simply, but complex software designs which fulfil their requirements are not so simple to develop.

²Many of them also coded.

- Software designers work in a highly dynamic environment. As hardware capabilities improve and software requirements grow, the tools which designers are expected to use become increasingly more powerful (and complex). Designers are faced with a dilemma. They can either:
 - Stick with one method, become familiar with it and work within its limitations.
 - Continually evolve their methods to cope with all the new research on software development.
 - Change methods and techniques to suit the problem at hand.

Each of these approaches has its advantages and disadvantages. Clearly, no matter which choice is made, designers cannot be oblivious of the dynamic nature of their working environment.

5.1.4.3 Software Design: Help Is At Hand

Software designers have two advantages over other types of designers. Firstly, software is pliable and can be manipulated so much more easily than other more concrete designs. Secondly, many of the other problems facing software designers have been faced by other types of designers in a wide range of problem domains. Software design is a new discipline, but many of the same principles and techniques used in other design areas are applicable in software development.

Design, in general, is about understanding requirements, understanding solution space, transforming structure and verifying design against requirements. A design method helps to co-ordinate these activities.

5.2 Learning From Different Design Areas

5.2.1 Allowing For Change: A Unique Problem

Software design is the only discipline in which designers expect the requirements they are given to continually change. In other areas, designers may be asked to extend their designs to incorporate new requirements, but only in computing are designers regularly expected to change their designs to accommodate requirements alterations. In other areas, designers consider such changes a major problem. In software design, such changes are the norm.

Perhaps wrongly, software engineering is seen as being inherently flexible: it is all too easy to change a few lines of code! This is true, but it is not easy to control the changes and to understand their consequences. The extremely pliable nature of software is both an advantage and a disadvantage. The dynamic nature of software requirements is a persistent problem. Changes in requirements occur during the design process, through the coding, and after the “final” product has been completed. Correcting mistakes and extending the software, otherwise known as *maintenance*, is an unending process. It is the compliancy of software which makes this possible. Problems arise because this is adversely taken advantage of by programmers. Complex software systems are prone to being made incomprehensible by uncontrolled change. Designers can, and should, play an important role in preventing this from happening.

The object oriented development strategy advocated in this thesis enforces controlled change. Designers should not be given a system (or parts of a system) to design until the requirements are fully understood and unlikely to change. In this way designers can work independent of analysts. Certainly mistakes will be made, but these are the analysts' responsibility, not the designers. The correction of mistakes in the requirements model should be easy to filter through to the design, provided the initial transformation to design maintains a mapping between components, and the subsequent design decisions are well documented. Similarly, if the requirements model is defined with modifiability and extendibility in mind, the resulting designs should also exhibit these features (to some degree).

5.2.2 Identification of General Techniques and Principles

The central theme of design is structure management. Like analysts, designers manage complexity by enforcing structure on the way in which behaviour is represented. (The difference between these two development stages is that analysts work solely with problem domain structure, whilst designers work with structure which, at each stage of design, is a fine balance between problem domain and solution domain architectures). Consequently, some of the techniques and principles evident in analysis are also evident in design:

- The importance of notation (language of expression).
- The importance of structure: hierarchical and configurational.
- The role of re-use.
- The ability to test a model against requirements.

5.2.2.1 Design Language

Design is concerned with communication. Consequently, the design language is fundamental to the design process. Language is any means of communication through the use of conventional symbols. Everyone is familiar with their own natural language. What is surprising is the number of other languages from which people can acquire some information: for example, maps of all various types, architectural plans, furniture construction instructions, mathematical equations, chemical formulae, pages of music, chess notation, recipes, etc

Certainly there will always be a relationship between natural language and other forms of representation since natural language shapes the way in which we can think. There is always a good reason why natural language is not used to communicate certain types of information:

- Natural language is not good for communicating spatial properties.
- Natural language is too expressive and often what is required is a simpler notation.
- Natural language is open to interpretation.

Languages are developed to make the recording of certain information simple, elegant and concise, whilst making the representation of other information very difficult. Language is the most important tool for abstracting away from unimportant information. Abstraction is fundamental to all areas of design.

5.2.2.2 Structure: (de)composition

The doctrine “divide and conquer” is central to all human activities and can be applied particularly well to design processes. Targetted design involves taking a set of requirements and producing an object which can be said, in some way, to fulfil these requirements. Designers decompose a requirements model to achieve a structured understanding. All designers follow (or advocate in part) repeated application of the “divide and conquer” maxim. Structured requirements aid the process of decomposition by providing a natural means of initially decomposing a problem. Design is also about composition: putting components together to form new structures applicable to the solution domain.

5.2.2.3 Re-use

All designers re-use elements of their working environment: experience, methods, structures and components. Software designers must learn from other areas in which re-use is prominent. This learning can be passed on to a community of designers when it is incorporated in a general design method. The main form of re-use advocated for the object oriented design stage of FOOD is the repeated application of correctness preserving transformations. Designers are then re-using well defined ways of taking designs from the abstract to the concrete. Component re-use is more prominent in the analysis and implementation stage of FOOD.

5.2.2.4 Testability

All designs must be tested against requirements. In many design areas these tests are informal and difficult to guarantee. Software designs are very difficult to test because the requirements which they are developed to fulfil are usually very complex. Rapid prototyping and modelling are well accepted ways of testing. This thesis shows that formal object oriented design is particularly well suited to rapid-prototyping.

5.2.3 Software Design and Engineering

Software design is often called software engineering. This is a reflection of the similarities between the roles of software designers and engineers of all disciplines. Engineering, in general, has well established methods which are governed by physical laws. Engineers learn to employ standard means of representation. Systems being engineered can, in general, have their approximate behaviour determined through analysis of the design documentation. Standard mechanisms and tools exist for constructing solutions from many types of engineered design.

Rather than expanding on the engineering analogy, this thesis acknowledges that software development should be extended to reflect the practices evident in engineering disciplines. In particular, formal techniques of software development are lacking in development method. *Formal methods* are really a set of models and tools which are usually distinct from a particular method (way of using the models and tools). Engineering balances method with the underlying models (based on physical laws and mathematical systems) in a way which software designers should attempt to emulate.

5.3 Object Oriented Software Design

5.3.1 Overview of Software Design

Rather than reviewing a wide range of particular software design methods, whose main role is to go from problem domain structure to solution domain structure, this section borrows an approach by Meyer [84] by identifying criteria for evaluating design methods and stating the principles upon which good software design should be based.

5.3.3.1 Design Criteria

The criteria for judging design methods are similar to the criteria for judging analysis methods (see chapter 2):

- Design languages must incorporate explicit structuring mechanisms.
- Design methods must encourage a structuring in which there are components to match elements in both the problem domain semantics and solution domain semantics.
- Design methods and models must combine in a consistent and coherent fashion.
- Design methods must be flexible to allow for designer creativity.
- Design methods must facilitate re-use and encourage the rapid development of experience.

5.3.3.2 Design Principles

When communicating complex ideas, it is important to keep things as simple as possible. The underlying principle is therefore to make the final design only as complex as the requirements demand, and to make the process of achieving this design as simple as possible to understand.

Most software development methods recommend the following as a means of reducing complexity:

- Strong cohesion.
- Weak coupling.
- Well defined interfaces between components.
- Encapsulation of components.
- Limiting the number of components at each level of abstraction.

Re-use is also a prominent feature of design. It is argued that re-use aids understanding. The different types of re-use in software engineering are well documented: [59] provides a good overview of the subject. This thesis advocates re-use at all stages of software development.

5.3.2 Comparing Object Oriented Design and Object Oriented Analysis.

We have argued that object oriented development is superior to other development methods because of the conceptual integrity between problem domain and solution domain. In particular, we have stated that the problem domain structure should be present in the design. This begs the question:

what does an object oriented designer do if the structure is maintained throughout the whole object oriented development method? In answer to this question, we identify four responsibilities of an object oriented designer:

- The removal of nondeterminism.
- The concrete realisation of the abstract object oriented concepts as specified in the requirements model.
- The restructuring of requirements to suit an implementation environment.
- Verification of design against requirements.

Each of these responsibilities are examined in the following sections: 5.3.3 to 5.3.5.

5.3.3 Removing Nondeterminism

In chapter 5, two types of nondeterminism arise from the specification of probabilistic behaviour and implementation freedom in the requirements model. Designers must remove both types.

Removing probabilistic nondeterminism involves specifying the probabilistic requirements in some standard way which is amenable to immediate coding. This type of design step is not examined in any detail in this thesis. The analysis method in chapter 4 identifies a means of recording the probabilistic behaviour which separates probabilistic properties from other behavioural concerns. Consequently, these less-abstract properties can be abstracted away from during design. It is beyond the scope of this thesis to investigate the design and implementation of probabilistic behaviour.

Removing the nondeterminism due to implementation freedom in the requirements is one of the major responsibilities of design. Analysts are encouraged to offer sets of alternate permissible behaviours from which a designer is required to choose one particular solution. One way of removing this type of nondeterminism is specified by the *Rend CPT*, defined in section 5.7.4.

5.3.4 Realising the Abstract Object Oriented Model

The object oriented requirements model must not specify implementation concerns. An analyst is not concerned with whether the objects are going to be implemented as, for example, concurrent processes (on distinct processors), imperative records or Eiffel class instances. At the simplest level, the analysis model does not even state how objects communicate with each other, or how their state is realised. The designer must know the way in which the abstract semantics in the requirements model can be mapped on to the target implementation language semantics.

5.3.5 Restructuring The Requirements To Match An Implementation Environment

Restructuring has two main goals:

- To facilitate re-use of implementation code.
- To take advantage of the high level language constructs in the implementation language.

5.3.5.1 Restructuring For Re-use

A flavour of the designers role in the process of restructuring for re-use is illustrated by the two examples in figures 5.1 and 5.2.

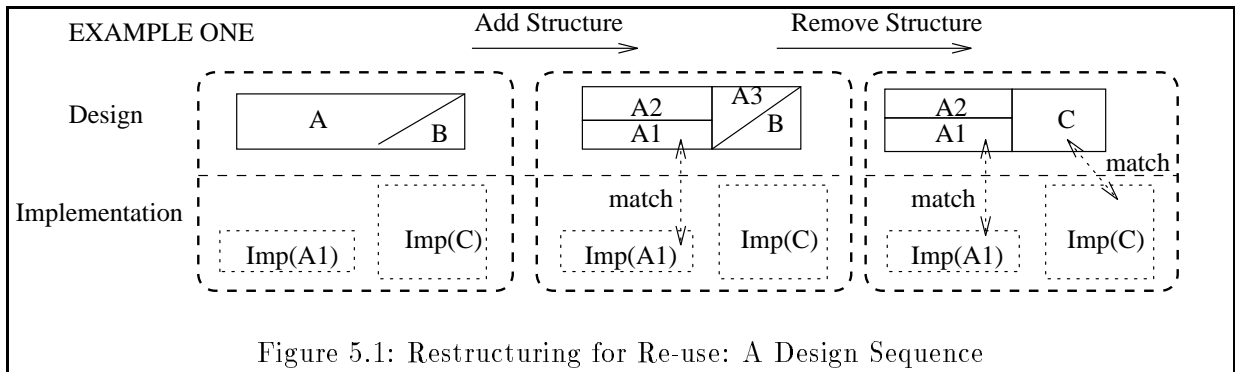


Figure 5.1: Restructuring for Re-use: A Design Sequence

In example one, in figure 5.1, the designer has identified two re-usable implementation components which provide part of the behaviour in the first design. Implementation component **A1** provides part of the behaviour of design component **A**. Implementation component **C** provides all of the behaviour of design component **B**, together with some of the behaviour of **A**. The designer can restructure the design to utilise the already existing components, and make the job easier for the coders. The first design step is to decompose **A** (i.e. add structure) so that one of the design subcomponents matches the implementation component **A1**. In this step the designer also structures the other subcomponent into parts (**A2** and **A3**) in anticipation of the next composition step. The next design step is to compose design components **A3** and **B** to make a match with implementation component **C**. At this point of the design, the implementers must code only one new component, an implementation of **A2**, and combine three components, namely **A1**, **A2** and **C**.

Example two, in figure 5.2, represents a branching in the design.

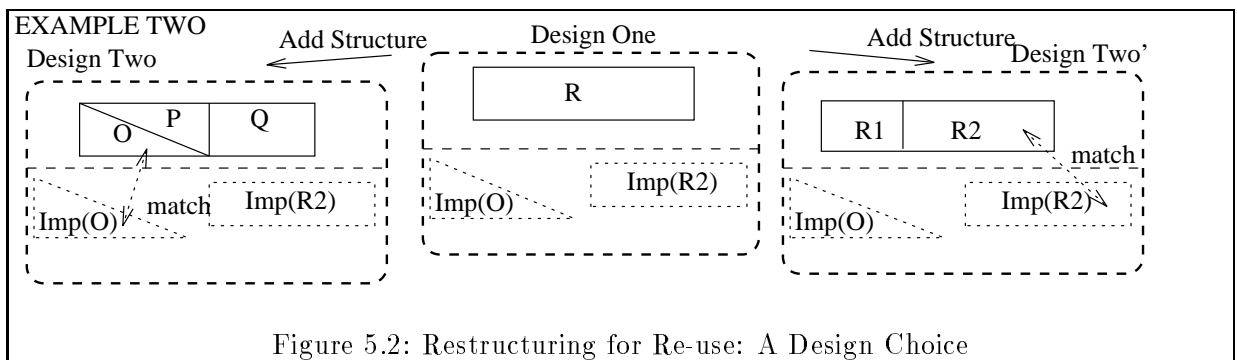


Figure 5.2: Restructuring for Re-use: A Design Choice

The designers are aware of two components which each provide a part of the behaviour specified in the design. However, the two components cannot be used together (perhaps because of some constraints in the implementation environment). The designers would like to re-use at least one of them, and so must choose which design trajectory to follow (both involve adding structure).

Example two also illustrates a more complex design trajectory. Consider **Design Two** as the initial

design. A ‘small’ part of the design, namely $\mathbf{0}$, is directly implementable (i.e. a piece of pre-written code exists). However, a different implementation component, represented as $\mathbf{Imp(R2)}$, provides a large amount of behaviour of the design components $\mathbf{0}$, \mathbf{P} and \mathbf{Q} . The designer ignores the obvious reuse of component $\mathbf{0}$ and restructures the design to finish with stage **Two**’ (via stage **One**). These types of restructuring design steps are facilitated by the correctness preserving transformations defined in section 5.7.1. and 5.7.2.

5.3.5.2 Restructuring For Implementation Language Utilisation

The implementation language which the designer is aiming towards may offer high level constructs which the designer must attempt to utilise. The implementation environment may offer, for example, concurrency, distributed processing, multiway synchronisation or sharing. Designers must structure their designs to make these properties explicit in the design, so that there is a better match with components in the implementation language. The correctness preserving transformation *Dist*, specified in section 5.7.3, performs this type of role.

5.3.6 Verification and Correctness Preserving Transformations

5.3.6.1 The Need For Formality

The most important role of the designer is to verify that the designs produced fulfil the requirements. Formal requirements models and formal designs are essential in improving the verification process, and making the customer (and software producer) confident in the final product. Given a formal requirements model, the general verification of any given formal design against the requirements is very difficult and, except in the most simple circumstances, impossible to guarantee in the present development environment. However, all is not in vain. Designers can be encouraged to follow a design trajectory in which the design evolves in a number of stages. Each stage can then be verified against the original requirements model.

The design trajectory method works for the following reason. The first design can be verified against the requirements. Then this design can be manipulated to achieve the next design, and so forth. Provided each design is verified against the previous one, by induction, the final design is verified against the original requirements. (Note that the correctness relationship must be transitive.)

Following a design trajectory does not make the verification process at each stage of the design any easier. For example, a designer can make any number of complex changes between each stage. In such cases, designers are back to the original problem: verify any given design against the requirements model. Consequently, most design trajectories restrict the type of changes that can be made at each design stage.

Correctness preserving transformations (CPTs) take this restriction one step further. Designers which restrict themselves to using CPTs do not need to do any verification at all! The CPTs are defined, and proven, to guarantee that, if a design, $D1$ say, is transformed by a CPT into another design, $D2$ say, then some properties of $D1$ are guaranteed to be fulfilled by $D2$.

5.3.6.2 The Advantages of an Object Oriented Environment

The specification of CPTs is not easy. Designs go from the abstract to the concrete. CPT development, in an object oriented framework, is aided by consistency in the underlying semantic model at all levels of abstraction. A major difficulty is in identifying transformations which are used throughout the design process, and formally defining them as CPTs. Another difficulty in defining such a design trajectory is in verifying the first design against the requirements model, especially if the semantics of the two models are very different.

The approach taken in this thesis means that the original requirements model is directly incorporated in the first design. The ACT ONE model of the object oriented requirements is carried over to the initial full LOTOS design specification. Further, the CPTs are defined only on the object oriented LOTOS specifications which, by definition, contain the original ADT requirements model. This makes the specification of CPTs much easier than in the general case, where they must be defined on the domain of all valid LOTOS specifications.

5.4 Object Oriented Design with LOTOS

5.4.1 Design in LOTOS

5.4.1.1 An Overview Of Design and Verification

LOTOS may not appear, at first glance, entirely appropriate for formal software design. It was developed for use in the area of standards (particularly service and protocol specification) and consequently its semantics is abstract: the specification of standards must be implementation independent so that manufacturers are not restricted in the way they can develop products to fulfil the standards. Manufacturers must permit testing of their products against standards, without having to give access to internal details. Thus, most conformance relations in LOTOS have been defined to be observational [10]. Verification of LOTOS designs is not restricted to ‘black box’ testing. Designers have complete views of the designs before and after each design stage. Thus, the conformance relationships used by designers need not be restricted to being observational in nature.

5.4.1.2 Design Is About Structure

Inherent in the design process is the notion of structure. Design is the process of creating a framework upon which a set of requirements can be realised. Design languages must have explicit structuring mechanisms. An advantage of software design is that the requirements are (formally) specified and as such will be structured. This gives designers an initial framework from which they can gain understanding of the requirements, and on which they can start to create a design. An object oriented requirements model is even more advantageous since the requirements model structure is likely to provide a good framework on which to start the production of an object oriented implementation.

There are two types of structure which are fundamental to design:

- **Hierarchical**

Hierarchical structure is one in which elements of the design are related by an ordering. In object oriented design, subclassing between classes and composition between objects define two different types of hierarchical relationship.

- **Configurational**

Configurational structure is one in which elements of the design are connected by relationships which do not have an implicit ordering.

LOTOS provides facility for defining both types of structure property. Consequently, LOTOS fulfils one of the necessary (but not necessarily sufficient) conditions for a language to be suitable for software design: it must support structured specifications.

5.4.1.3 Software Design Is About Controlling Change

Another aspect of software design is that the designs must be manipulatable. Designs must be amenable to change so that a design trajectory is simple to follow. This is certainly true of LOTOS, but pliability is only half the story. The manipulations must be controllable: the reasons for making a design change must be fully understood, and the consequences deterministic.

All structured design methods (including object oriented approaches) argue that their techniques are advantageous because changes can be kept as local as possible. Often they identify the need only to change one component of the design at a time. Certainly LOTOS offers this type of local change facility: ACT ONE sorts and LOTOS processes can be treated as modular elements. However, design is not just about localising change. For example, a high level design decision might be to change the server-client communication model throughout the whole design (in response to a change in the target implementation language, for example). Such a change is inherently global. The use of CPTs helps to make such changes in a controlled way. LOTOS specifications are formal and are therefore amenable to controlled global change.

A final requirement of a design language, with respect to structure and structure manipulation, is that it can express behaviour at very different levels of abstraction. The initial design must be very close to the requirements model, whilst the final design must be very close to an implementation model. A language which can represent a range of behaviours, from abstract to concrete, is called a wide-spectrum language. Software design languages which are used throughout a comprehensive design trajectory must be wide-spectrum. LOTOS is such a language.

5.4.1.4 Designs Must Be Verifiable

Designs must be verifiable against requirements. LOTOS, as a formal language, is open to mathematical verification against a formal requirements model. However, formality alone is not sufficient. Formal methods are dependent on tool support. It is impossible to verify even the simplest set of requirements by hand. Consequently, we require that a formal design language must have a reasonable tool support, with further support in the foreseeable future. LOTOS fulfils this requirement.

5.4.2 Abstract Data Typing in LOTOS

LOTOS contains an abstract data typing language based on ACT ONE. Chapters 2 to 4 illustrate the power of ADTs to express structural properties (albeit in an object oriented framework of interpretation). Chapter 4 also identifies the weakness of the ADT approach to modelling: ADTs are not good for recording communication, synchronisation, timing, concurrency and distribution properties. Quite deliberately, in this thesis, these high-level design features are abstracted away from during requirements capture. The requirements model says *what* rather than *how*. ADTs do play a major role in LOTOS designs: they maintain the underlying abstract behaviour whilst the process algebra is used to define the more concrete high-level design properties.

5.4.3 The Process Algebra in LOTOS

LOTOS is also constructed from a process algebra. These languages can, by themselves, record requirements in a highly structured fashion, which can then be interpreted as high-level designs. For example, processes can be decomposed into component processes which combine (using the parallel operators). This type of decomposition gives rise to hierarchical and configurational relationships structure. The components' interaction during event synchronisation is configurational, whilst the (de)composition is hierarchical. There has also been much work in defining hierarchical relationships between processes, based on the behaviour they offer. For example, there are many inter-process relationships which attempt to model subclassing and implementation properties, see for example [9, 33, 8].

Given the structural expressiveness of the process algebra part of LOTOS alone, we must question why the ADT part is required. The ACT ONE is necessary for defining (or modelling) the following:

- Parameterised behaviours in the shape of parameterised process definitions.
- Systems with explicit state components, also in the form of parameterised processes.
- Non-constructive properties in the form of preconditioned (guarded) behaviour.
- Structured events, which are necessary to model value matching, value passing and value generation.
- Process functionality.

LOTOS without the abstract data typing, often called basic LOTOS³, can specify only a limited range of behaviours. This thesis uses the ACT ONE specification, as it is generated from the object oriented requirements, in conjunction with the process algebra.

5.4.4 Balancing Processes and Types in Design

One of the main problems with designing in LOTOS is in achieving the correct balance between ADT specification and process algebra specification. This thesis advocates using the ADT part as

³The terms 'basic LOTOS' and 'full LOTOS' are used to distinguish between LOTOS specifications with and without ADT parts, respectively.

a functional behaviour carrier and the process algebra as a high level structuring mechanism for specifying communication, synchronisation, concurrency, etc. . . . Generally, in LOTOS specifications, the balance between the two parts of the language is not so clearly defined.

A good introduction to LOTOS specification and the different roles of the process algebra and ADT parts is given in [120]. It also introduces the notion of specification style, and the way in which different specification styles place different emphasis on the roles of each part of the language. In general, different styles are best suited to specification at different levels of abstraction. Consequently, one approach to design is to specify a number of transformations between LOTOS styles. However, it is not yet possible to automate the whole design trajectory in the form of a complete set of CPTs. Rather, designers will be expected to directly interact with, and perform manipulations on, the LOTOS designs. Designers should not be asked to cope with LOTOS specifications written in many different styles. Jumping between different conceptual frameworks does not aid the design process.

This thesis advocates a design trajectory in which the style of LOTOS specification remains consistent. The specifications progress from the abstract to the concrete, but the underlying object oriented conceptual framework is maintained. It is the balance between the amount of behaviour specified in ACT ONE , and the amount specified in the process algebra which changes as the design evolves. In this way there is a clear reasoning behind the balance at any particular point in the design process.

LOTOS specifiers often have their own preferences in the way in which they use the ADT and process algebra parts. This favouritism is probably a consequence of their familiarisation with, and understanding of, the two different types of semantics underlying the two languages. It is not good that specification designers can influence the structure of their designs in a way which is not amenable to analysis. Another reason why the balancing between ADT and process algebra parts is so subjective is that there are no well accepted methods for developing formal specifications in LOTOS. There are plenty of tools for automation, validation and verification, and a wide range of example specifications, but there is little advice (and tool support) for the actual process of constructing the specifications. In particular, there is a real lack of management support⁴. Consequently, there are no existing methods (or tool support) for combining the ADT and process parts in a consistent and coherent way. Both parts can be used to record structured information but, without a method (or guidelines, at the very least), it is difficult to say which types of behaviour should be defined using which part.

5.4.5 Defining an Object Oriented LOTOS Style of Specification

Many attempts have been made to define object oriented (or object based) styles in LOTOS, for example [118, 100, 81, 24, 35, 6]. There is a vague consensus of understanding concerning the mapping between LOTOS constructs and the object oriented paradigm:

- Processes define classes of behaviour, usually of type `noexit`.
- Objects are instances of processes. The state of an object is represented by the parameterisation of the process.

⁴Perhaps this is the real reason why formal methods, like LOTOS, have not become accepted in industry?

- Objects (process instances) service requests through interaction at their external gates.
- The passing of parameters (input and output) between client and server corresponds to event synchronisation (and value agreement) at these gates.
- Subclassing is some relationship between process instances (objects).

There are two problems with this informal correspondence:

- The ADT part does not seem to have an active role.
- There is no relationship between classes of behaviour: the parameterised process definitions. The behavioural relationships are defined between process instances, and thus there can be confusion in differentiating between object and class.

In the object oriented LOTOS designs developed in this thesis, the ADTs play a very important role: they maintain the behaviour specified in the requirements model.

In object oriented design, there needs to be a clear distinction between classes and objects. When using LOTOS, confusion arises because process instances define a set of behaviours and this gives the impression that such a set is a class. This thesis argues differently. In our object oriented interpretation of LOTOS, each process definition corresponds to a class. Process instances correspond to objects and the set of behaviours defined by each object represents the set of valid implementations. The ADTs define the underlying behaviour of each class and the process algebra part of each class definition defines the high-level properties of the system.

The fact that one standard object oriented style of LOTOS specification has not been formally defined and well-accepted is indicative of the problems in the object oriented community. This thesis argues that the problem is not with LOTOS, it is with the inherent informality in object oriented systems and the many different interpretations of the object oriented concepts.

Object oriented concepts are not well understood, although there has been some recent work in defining object oriented semantics [47, 95, 129, 37, 29, 130]. These semantics were not chosen for use in this thesis because:

- They do not take a natural *state-transition-system* view of objects and classes.
- They do not recognise the importance of an object offering a constant interface during its lifetime.
- They do not match our intuition of objects and classes at all stages of software development.

Defining an object oriented semantics in LOTOS appears, at first glance, to be a rather appealing solution to the problem of informality in object oriented designs. However, such a solution is not general enough since the object oriented model so produced is too concrete for use during analysis. For example, even something as simple as the client-server communication model cannot be specified in the process algebra part of LOTOS without straying into implementation details. Such a model inherently restricts object oriented implementers to: synchronous or asynchronous communication, concurrency or sequentiality, distributed or centralised control etc. . . . Full LOTOS is a good language to specify object oriented models at a concrete level, but using the process algebra during analysis and requirements capture may be too soon.

5.5 FOOA as Input to Formal Object Oriented Design

5.5.1 Generating Full LOTOS from the Requirements Model

The requirements model, as specified in OO ACT ONE, and realised in the translation to ACT ONE, is an abstract statement of the behaviour a system is required to offer. The object oriented requirements say *what* is required rather than *how* a solution should be implemented. This is illustrated very well when the initial mapping to full LOTOS is considered.

Given a set of requirements of a system specification in an OO ACT ONE class definition, there are a number of ways in which these requirements can be translated to an initial abstract LOTOS design specification. Common to all such translations must be the retention of the ACT ONE class definitions (as represented in ACT ONE) in the full LOTOS code. This makes verification of the initial design against the requirements model straightforward.

Object oriented designers must initially identify the communication aspects of the way in which the underlying object oriented behaviour is to be fulfilled. The designers of a system must decide how the behaviour is to be offered at its external interface (and what this external interface should look like). This simple decision can affect the rest of the design process. Identifying an object oriented communication model and specifying the translation from OO ACT ONE, is not simple. There are a number of alternative models and a number of ways in which these can be specified. Four of these alternatives are examined in the following sections (5.5.1.1 to 5.5.1.4). The list is not exhaustive and the ways of specifying the models are limitless.

5.5.1.1 Remote Procedure Call (RPC) Model

The RPC model is based on the principle that while an object is servicing a request, no more requests can be accepted. Consider such a specification for the well accepted **Stack** behaviour⁵. The **Stack** elements are arbitrarily chosen to be **Nats**. This same **Stack** behaviour is also used to illustrate the other initial design alternatives. The RPC **Stack** behaviour is defined in the **RPCStack** process, below.

```

RPC: Stack example one
process RPCStack[push,pop](SStack: Stack): noexit:=
(push? Nat1: Nat; RPCStack[...](.(push(SStack, Nat1))))
[]
(pop; pop! NatResult(pop(SStack)); RPCStack[...](.(pop(SStack))))
endproc (* RPCStack *)

```

This style of specification is useful when the target implementation language has a procedural communication/interaction semantics. The **RPCStack** clients must wait for the **Stack** object (**RPCStack** process instance) to finish servicing its current request before their requests are accepted. In effect,

⁵In this example, and all others that follow, non standard syntax for the specification of the process gate list is used. When a gate list in a process instance is to be specified exactly as the gate list in the process header then it is more concisely written as [...].

the `RPCStack` refuses to participate in service request events if it has not yet finished servicing its current request. Note that this *lock-out* does not occur when `TRANSFORMER` attributes are serviced.

5.5.1.2 Parallel Access Model (Ordered In)

In this communication model, an object can service all requests at any time in its life (i.e. there is no *lock-out*). The order in which the requests are serviced is the order in which they are requested. However, the order in which the replies are given back to the requesting environment may not be maintained. The LOTOS specification of the `Stack` behaviour in this communication framework is given in the `PAMStack` process definition, below.

Parallel Access Model (Ordered In): Stack Example 2

```

process PAMStack[push,pop](SStack: Stack): noexit:=
  (push?Nat1:Nat; PAMStack[...](. (push(SStack, Nat1))))
  []
  (pop; ((pop!NatResult(pop(SStack)); exit) ||| PAMStack[...](. (pop(SStack)))))
endproc (* PAMStack *)

```

The `PAMStack` process can always accept a `push` or `pop` request (i.e. participate in a `push` or `pop` event). The transformer attribute `push`, like in the RPC model, is served instantaneously: the resulting state transition (re-instantiation of the `PAMStack` process with new state parameter) is achieved without need for a sequence of internal events. The dual operator `pop` is defined in terms of two event synchronisations: the attribute request `pop` and the attribute response `pop!NatResult(pop(SStack))`. Unlike in the RPC model, the `PAMStack` process can accept other service requests between receiving a `pop` service request and returning the `pop` result. This is specified using the parallel operator (`|||`). The result of the `pop` request is offered in parallel with the behaviour of the `PAMStack`. A consequence of this communication model specification is that, since multiple results can be offered in parallel, results do not necessarily have to be `popped` off in the order in which they are requested. This type of property is, in general, undesirable. Consequently, we do not consider this model for use during FOOD.

5.5.1.3 Parallel (Ordered In Ordered Out) Model

In this model, an object can service all requests at any time. The order in which the requests are serviced is the order in which they arrive. The order in which replies are sent is also maintained by the serving object. The *Ordered In Ordered Out Stack* behaviour is specified by the `ParStack` process in example three.

This LOTOS specification is much more complex than the others. It is providing `Stack` behaviour wrapped between input and output queues. The `StackIn` and `StackOut` processes are parameterised on a `Nat`. These parameters are used to tag requests as they come in and guarantee the order of responses on the way out, respectively⁶ The queueing of service requests and responses is achieved by the parallel operators in the specification. There is no explicit queueing behaviour defined in the

⁶The state parameters of the `In` and `Out` processes can be initialised to any value provided it is common to both.

```

Parallel (Ordered In Ordered Out): Stack example three
process ParStack[push,pop](SStack: Stack): noexit:= hide request, response in
StackIn[push, pop, request] (0) |[request]|
StackBody [request, response](SStack) |[response]|
StackOut [pop, response](0) where
process StackIn[push, pop, request] (ID: Nat): noexit :=
Reqs[push,pop,request](ID) |[request]| ReqController[request](ID) where
process Reqs[push,pop,request](IDsStackIn:Nat): noexit:=
(push? Nat1:Nat;
 ( Reqs[push, pop, request] (.inc(IDsStackIn))) ||| (request!push!Nat1!IDsStackIn; exit)))
[]
(pop;
 (Reqs[push,pop,request](.inc(IDsStackIn)))||| (request!pop!IDsStackIn; exit)))
endproc (*Reqs*)
process ReqController[request](ServeID:Nat):noexit:=
(request!push?Nat1:Nat!ServeID; ReqController[request](.inc(ServeID)))
[]
(request!pop!ServeID; ReqController[request](.inc(ServeID)))
endproc (* ReqController *) endproc (*StackIn*)
process StackBody[request, response](SStack: Stack): noexit:=
( request!push? Nat1: Nat?ID:Nat;
 (StackBody[request, response](.push(SStack, Nat1)))||| (response!push!ID; exit)))
[]
( request!pop?ID:Nat;
 (StackBody[request,response](.pop(SStack)))||| (response!pop!NatResult(pop(SStack))!ID; exit)))
endproc (*StackBody*)
process StackOut[pop, response](CountStackOut: Nat): noexit:=
(response!pop?NatStackOut:Nat!CountStackOut;
 pop!NatStackOut; StackOut[pop, response](.inc(CountStackOut)))
[]
(response!push!CountStackOut;
 StackOut[pop, response](.inc(CountStackOut)))
endproc (* StackOut *) endproc (* ParStack *)

```

ADT. The translation to the parallel (ordered in ordered out) LOTOS model (for conciseness we call this the Par model of communication) requires an ACT ONE sort `Nat`, to provide the unique identification for each request, and an ACT ONE sort with literal members `push` and `pop`, which by convention is named `StackServiceRequests`, in order to differentiate between internal service requests. In translation, all classes in the OO ACT ONE specification have their external attributes defined as literals in a `ClassNameServiceRequests` sort. All these sorts are defined in a global `ServiceRequests` type specification.

Further, the `Nat` parameter sort can be replaced by any sort which offers a means of allocating an infinite set of unique identifications. `Nat` was chosen for its simplicity.

5.5.1.4 Parallel Explicit Routing Model

In this model, an object can service all requests at all times. The ordering of servicing is maintained, but the ordering of replies is not. However, each service is tagged, by the environment, with a unique identifier. This means that the environment can control the ordering and guarantee that the replies get returned to the correct clients. The explicit routing model of `Stack` behaviour is defined by the `ExpStack` process, below.

Explicit Routing: Stack example four

```

process ExpStack[push,pop](SStack: Stack): noexit:=
(push? Nat1: Nat? ID: Nat; ExpStack[...](.(push(SStack, Nat1))))
[]
(pop?ID:Nat; (pop!NatResult(pop(SStack))!ID; exit) ||| ExpStack[...](.(pop(SStack))))
endproc (* ExpStack *)

```

This specification is similar to the `Stack` behaviour defined in example two. The only difference is that the requests are accompanied by an identification (ID) which must be provided by the client of the `Stack`. These IDs are then tagged to the `pop` replies. This type of communication model can be utilised in the design process to define internal communication. However, explicit routing is, in general, too concrete a model to be used in the initial design stages. Consequently, we do not consider it for use in FOOD.

5.5.2 Internal and External Communication

The four models, above, define external communication properties for classes of objects. Two of these, namely the RPC and Par models, are used to define two fundamentally different communication models. These models define only the external interaction between a client and a server. They do not specify the internal communication which occurs when an object is servicing a request. The reason for this is simple: there is no internal communication in the RPC and Par processes. These processes are not defined as interacting systems of component processes (i.e. they are unstructured⁷). In section 5.7 we examine a means of structuring LOTOS designs. The means of interaction between component processes of a system is said to be defined by the resulting internal communication model.

5.5.3 Defining the Mappings from OO ACT ONE to Full LOTOS

Given an OO ACT ONE class specification, `class` say, then we define two transformations for the generation of full LOTOS specifications of `class` behaviour:

- `MakeRPC(class)` produces the RPC LOTOS design of `class` behaviour in a `RPCclass` process definition.
- `MakePar(class)` produces the Par LOTOS design of `class` behaviour in a `ParClass` process definition.

⁷More precisely, all the composition structure is contained within the ADT part of the design.

Appendix E2 defines these two mappings. Central to each mapping is the inclusion of the ACT ONE requirements model of `class` behaviour.

In the remainder of this thesis, any process identified as `RPCclassname` or `Parclassname` is assumed to have been derived from the OO ACT ONE specification of `classname` using the appropriate mapping.

5.5.4 An Object Oriented Interpretation of the Initial LOTOS Designs

5.5.4.1 Notation Conventions

The OOLOTOS⁸ specifications follow the following syntactic conventions:

- Every class in the OO ACT ONE system requirements has an ACT ONE sort of the same name. These sorts are specified in the ACT ONE requirements model generated from the OO ACT ONE specification.
- An instantiation of the `system` class corresponds to an instantiation of an RPC or Par process.
- The gate list of the `system` class process corresponds to the list of transformer, accessor and dual attributes of the class. (The ordering in the OO ACT ONE specification is maintained in the LOTOS code.)
- The state of the `system` class process is identified by the variable `Ssystem`, a value of the ACT ONE sort `system`.

5.5.4.2 Processes and Objects

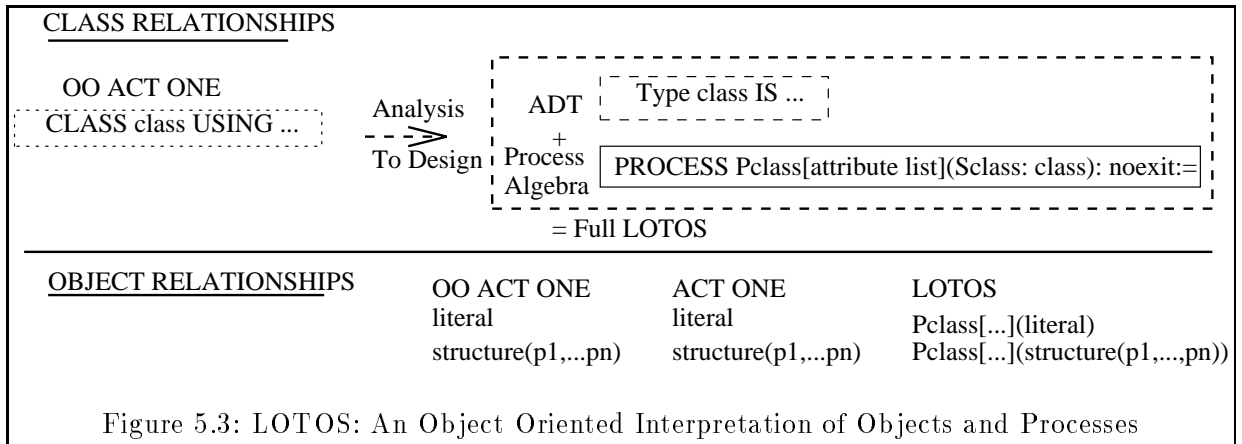
The relationship between OO ACT ONE classes, ACT ONE sorts, and LOTOS processes is illustrated in the top half of figure 5.3. The internal aspects of the process specifications expand out as design progresses: they are not represented in the figure. The structure of the problem domain is retained in the ADT part of the LOTOS designs. The CPTs can be used to transfer this structure to the process algebra for further manipulation. The bottom half of figure 5.3 shows the simple relationship between objects in the three different notations.

5.5.4.3 External Attributes and Servicing Requests

There is a direct correspondence between the gates of a process and the external attributes of the class which it is modelling. The hidden attributes of a class are not included as part of the external gate list of the class process and, consequently, the behaviour offered by the hidden attributes can be accessed only through the ADT specifications. The internal (nondeterministic) transitions in the requirements are modelled as internal events by the LOTOS `hide` operation.

The parameterisation of the external attributes is matched by the event synchronisations between the process and its environment. A service request is modelled, in the server process, as an ‘input

⁸OOLOTOS specifications are defined to be those specifications which are derived from OO ACT ONE requirements models.



event' of the form: `attribute?p1:P1?...?pn:Pn`, where there is an external attribute, `attribute`, defined to have class parameters `P1, ..., Pn`. The result of an accessor or dual attribute is modelled as an 'output event' of the form: `accdual!result`, where `result` is a value of the appropriate ACT ONE sort. These correspondences are shown, for `Stack` behaviour, in figure 5.4.

	TRANSFORMER	DUAL/ACCESSOR
OO ACT ONE	<code>empty.push = S(empty,0)</code>	<code>empty.pop = empty AND ~Nat</code>
ACT ONE	<code>.(push(empty,0) = S(empty,0)</code>	<code>pop(empty) = dualStackNat(empty, unspecNat)</code>
LOTOS	<code>(PStack[push,pop](empty) [push,pop] (push!0; Q[push,pop])) = push!0; (Q[push,pop] [push,pop] PStack[push,pop](S(empty,0)))</code>	<code>(PStack[push,pop](empty) [push,pop] (pop; Q[push,pop]) = pop; (Q[push,pop] [push,pop] (PStack[push,pop](empty) (pop!unspecNat;exit)))</code>

Figure 5.4: LOTOS: An Object Oriented Interpretation of Service Requests

5.5.4.4 Composition

The object oriented composition properties of the initial designs are contained in the ACT ONE part of the full LOTOS specification. The compositional properties can be derived from the ACT ONE code, or from examination of the OO ACT ONE specification (and associated diagrams). During the design process, expansion transformations (see *StExp* in 5.7.1, for example) facilitate the internal decomposition of a given unstructured process into a system of component processes running in parallel. The configuration and communication aspects of component interaction are made concrete by the design CPTs.

5.5.4.5 Subclassing In Design

An obvious question is whether the subclassing relationships in the requirements models are somehow maintained across the transformation. More formally, given OO ACT ONE classes, A and B say, such that $A \sqsubseteq B$ and a transformation function, T say, which maps OO ACT ONE classes to LOTOS processes, is there any relationship between processes $T(A)$ and $T(B)$? In OO ACT ONE, only two subclassing relations are defined, namely extension and specialisation. Consequently, it is necessary only to investigate how these two relationships are carried across the analysis to design translation.

A Design Extension

The **Stack** behaviour is extended below for the RPC communication model. The extension is a **size** attribute which returns the number of elements currently on the stack. The new class of behaviour, which offers this additional attribute, is named **XStack**. The RPC model of **XStack** behaviour is defined below.

```

process RPCXStack[push,pop,size](SXStack: XStack): noexit:=
(push? Nat1: Nat; RPCXStack[..](.(push(SXStack, Nat1))))
[]
(pop; pop! NatResult(pop(SXStack)); RPCXStack[..](.(pop(SXStack))))
[]
(size; size!NatResult(size(SXStack)); RPCXStack[..](SXStack))
endproc (* RPCStack *)

```

In the RPC model of communication, it is evident that corresponding⁹ instances of processes **XStack** and **Stack** are not related by any of the standard testing equivalences [9, 10]: the **RPCStack** always deadlocks on event **size** whilst the **RPCXStack** does not. However, without going into formal details, the two process instances are related by a standard implementation relation, as defined in [16]: in a system containing an instance of **RPCStack**, the **RPCStack** can be replaced by the corresponding **RPCXStack** without the non-deadlocking behaviour of the system being compromised.

Consider the extension as it is carried across the *MakePar* mapping from analysis to design. The **ParXStack** code resulting from the *MakePar* mapping is given in Appx E1. The implementation relationship holds between **ParStack** process instances and **ParXStack** process instances.

A Design Specialisation

It is not possible to specialise the **Stack** behaviour since it is **nonpartitionable**. Consider instead a lift moving mechanism. The OO ACT ONE defining the **SMove** ('specialised move') and **Move** class interfaces, used in this example, is given below.

Consider the translation of this behaviour to an initial RPC LOTOS specification. (The same arguments apply for this model as for the Par model.) By definition, **Move** \sqsubseteq **SMove** (since **Move spec SMove**). The LOTOS **RPCMove** and **RPCSMove** process classes are defined below.

⁹Corresponding instances of two processes are those instances with the same state representation.

```

CLASS SMove USING BOOL OPNS
LITERALS: up,down,stay
TRANSFORMERS: flip
ACCESSORS: goingup -> Bool
EQNS ...ENDCLASS (* SMove *)
CLASS Move SPECIALISES SMove TO OPNS LITERALS: up, down ENDCLASS (* Move *)

```

```

process RPCMove [flip, goingup ](SMove):noexit:=
(flip; RPCMove[..](.flip(SMove))) []
(goingup; goingup!BoolResult(goingup(SMove)); RPCMove[..](.goingup(SMove)))
endproc
process RPCSMove [flip, goingup ](SSMove):noexit:=
(flip; RPCSMove[..](.flip(SSMove))) []
(goingup; goingup!BoolResult(goingup(SSMove)); RPCSMove[..](.goingup(SSMove)))
endproc

```

The process algebra definitions for these two behaviours are identical, except in the naming of the processes and process parameters, and the typing of these parameters. It is immediate that corresponding instances of these processes are weak bisimulation equivalent (written \sim) as defined by [9]. In other words, $\text{PMove}[\dots](\text{up}) \sim \text{PSMove}[\dots](\text{up})$ and $\text{PMove}[\dots](\text{down}) \sim \text{PSMove}[\dots](\text{down})$. The process instance $\text{PMove}[\dots](\text{stay})$ has no correspondences to any of the instances of PMove . This is precisely what is meant by **specialisation** in the object oriented semantic framework.

Definition: Class Relationships

The notion of a relationship between process instances is naturally extended to the notion of a set of relationships between sets of process instances. In this way, the notion of a class relationship can be developed in LOTOS. In LOTOS one says that process instances are related by some well defined relation. In an object oriented LOTOS, based on OO ACT ONE, this relation must be extended to parameterised process definitions, which correspond to classes. Given a relation R , between LOTOS behaviour expressions, a class relationship $\text{Class}R$ is defined as:

$$\text{PROCESS } PX \text{ } \text{Class}R \text{ } \text{PROCESS } PY \Leftrightarrow \\ \forall x \text{ such that } x \text{ is a value expression of sort } X, \text{ then } PX[\dots](x)R PY[\dots](x).$$

5.5.4.6 Polymorphism in Design

The RPC and Par models of communication, as presented above, do not incorporate the notion of polymorphism in the process algebra parts of the design. The polymorphic properties are defined in the ADT part of the designs but problems arise if polymorphism is not incorporated in the process algebra. For example, consider the **Stack** behaviour. The **Stack** is defined to accept **Nats** as input parameters of the **push** operation. However, if **Nat** is defined to have a subclass, **evenNat** say, the **ParStack** process cannot synchronise on a **push!evenNat1:evenNat** event, even though **Stack1.push(evenNat1)** is well defined in the requirements (in the OO ACT ONE and ACT ONE

models).

ACT ONE does not incorporate inclusion polymorphism semantics, but the ACT ONE class models (sorts) are specified to accept subclass parameter values through use of coercion and operation overloading. A similar approach must be taken when transferring the polymorphic requirements to full LOTOS. The RPC and Par communication models need to be expanded to cope with subclass input parameters. This is easily done: for example, the `ParStack` process contains the code fragment:

```
push!evenNat1; ParStack[...](.(push(SStack, evenNattoNat(evenNat1))))).
```

The inclusion of polymorphism properties in the process algebra part of the LOTOS designs is necessary, but its presentation can be ignored. Rather than including all the subclass parameter options in our full LOTOS design listings, the polymorphic behaviour is not presented. The design transformations presented in this thesis do not affect the subclassing hierarchy and so it is not, at present, necessary to consider the polymorphism properties when designing. The RPC and Par models are defined to specify the required polymorphic behaviour, even though it is not presented in the remaining sections of this chapter.

5.5.5 An Object Oriented Style of LOTOS Specification

An important aspect of the object oriented LOTOS designs is the specification of the *MakeRPC* and *MakePar* mappings, which define the *style* of the RPC and Par processes. Appendix E2 defines these two mappings. The RPC and Par processes are unstructured. They form the basic building blocks in the OO LOTOS *style* of specification. Three other basic building blocks are defined as the result of applying CPTs to these process definitions:

- ERPC ('expanded' RPC) processes result from applying a static expansion CPT (*StExp* in 5.7.1) to RPC processes. This produces a structured definition of the required behaviour in which the behaviour is composed from a number of RPC component process (in parallel) under the control of a centralised process.
- EPar ('expanded' Par) processes are a result of applying *StExp* to Par processes. This CPT results in a structured definition of the required behaviour in which the specification is composed from a number of Par component process under the control of a centralised process.
- Dist ('distributed') processes are a result of applying the *Dist* CPT (see 5.7.3) to EPar processes. This transforms a structured system of Par processes which have centralised control into a structured system of self-controlled (no pun intended) processes.

The OO ACT ONE *style* of specification is one in which the system is defined as a Par, RPC, EPar, ERPC, or Dist process. Further all the classes in the system which have been expanded to process form must also be represented in one of these five ways. In effect, the *style* of specification is defined by the initial communication models (RPC and Par) and the design transformations which can be applied to classes specified using these models. Consequently, the *style* is dynamic: when CPTs become well accepted design mechanisms, then the resulting process definitions will become well accepted design components.

In FOOD, the limited number of base *style* components can be usefully represented in diagrammatic form. The graphical notation is illustrated in figure 5.5

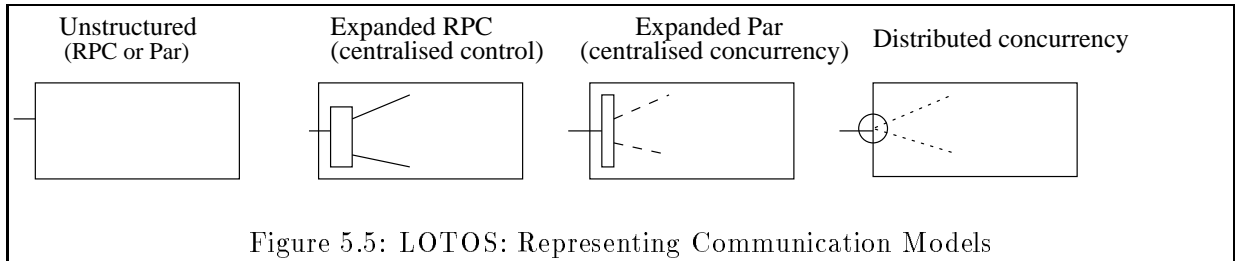


Figure 5.5: LOTOS: Representing Communication Models

5.6 Correctness Preserving Transformations (CPTs): Formalising Design

5.6.1 Introduction

This thesis examines the role CPTs can play in the process of design in general, and FOOD in particular. A transformation can be applied to a specification which reflects some architectural/implementation choice, without altering the external (observable) behaviour of the system. Such a design transformation is dependent on some nonstandard, though not necessarily informal, means of interpreting the internal details of the specification. The object oriented framework provides the basis for such an interpretation.

LOTOS, as a wide-spectrum language, can specify the properties of systems at various levels of abstraction. Design is the process which transforms an initially abstract (implementation independent) specification of system requirements into a final, more constructive, implementation oriented specification. An *ideal* LOTOS based software development environment should provide a comprehensive set of CPTs and a framework in which designers can apply these transformations to reflect design decisions. Such an ideal is a long way off. This thesis provides a small set of CPTs (in section 5.7) which are useful within our object oriented development method. These transformations are used to:

- Illustrate the CPT concept.
- Show the importance of matching design needs with CPTs.
- Highlight the difficulties involved in proving the correctness of the design transformations.
- Emphasise the power of a CPT-driven approach to design.

The small set of transformations proposed in this thesis do not constitute a design method. However, they do show how such a method could be constructed.

5.6.2 Concepts

5.6.2.1 Design Trajectory

A fundamental notion in this work is design trajectory: a sequence of steps which take a problem oriented specification of requirements to an implementation oriented specification of a possible solution. Each step changes the previous specification in some way. The important thing is that something must also be preserved along this trajectory: the ‘correctness of the design’.

5.6.2.2 Design Verification

In theory, it is possible to verify the correctness of any given design step by mathematical means. In practise, the complete formal verification of most design steps is not possible because of combinatorial problems. In these cases, specifications are partly verified by simulation and testing.

This thesis has already identified the advantages of simulation and testing with regard to analysis models. The same arguments are true for design models. The design approach advocated in this work does not restrict all design changes to be made through application of CPTs. Consequently, there may be a need for alternative verification methods. Two different types of LOTOS tools have been developed to help in this respect. Firstly, there are a wide range of simulation and automation tools (see [117, 10, 8], for example). Secondly, and more importantly, tools have been developed towards deriving tests from given LOTOS specifications (for example, [125] explains the theory behind a means of deriving canonical testers for LOTOS specifications). We do not examine any of these mechanisms, but we do recognise their value in this, and future LOTOS design methods. Rather, FOOD concentrates on a different approach to design verification, namely the application of CPTs.

5.6.2.3 Correctness Preserving Transformations

A different means of verifying a design is to perform only transformations (design changes) whose correctness has already been proven. Before examination of particular CPTs, a brief overview of the terminology is useful:

A specification can be said to be *correct* if it fulfils some property. Assume a specification S , a transformation T and define $S' = T(S)$, i.e. S' is the result of applying T to S . T can be said to be correctness preserving with respect to the property P if $P(S) \Rightarrow P(S')$. In other words, the property P is preserved across the transformation T .

This type of formulation raises a number of interesting questions:

- What sort of properties can be usefully preserved?
- How can these properties be formalised?
- Over what domains should T operate?
- What is the difference between S and S' which makes T a useful transformation for applying during design?

- Can we specify appropriate transformations to correspond with decisions most commonly taken by designers in practice?

Before these questions are more rigorously examined, the concept of property is given a useful categorisation:

- **External Properties**

External properties are those which can be observed through interaction with a system at its external interface (in LOTOS the external interface of a process is defined by its gate set). External properties, said to be purely functional, are fulfilled by a standard semantic interpretation of the specification. These properties are concerned with *what* the system does rather than *how* it does it.

- **Internal Properties**

Internal properties are those which can be derived through examination of the text which specifies the system in question. They cannot be ‘extracted’ through interaction with the system interface alone. Formulation of these properties requires the definition of a non-standard interpretation of the specification. This interpretation is said to provide a **view** on the system.

This categorisation gives rise to the classification of two different types of CPT:

- **Structural CPTs**

A structure CPT does not change the external properties of a system in any way. There are no ways of distinguishing the design before and after transformation through interaction with their external interfaces alone. Structure transformations change only internal aspects of the system.

- **Functional CPTs**

A functional CPT changes the external properties of a system but guarantees some sort of conformance between the design before and after transformation. In other words, a functional transformation compromises some external properties but maintains others.

5.6.3 An Overview of CPTs in LOTOS

5.6.3.1 The CPT Problem

By differentiating between what should stay the same and what should be different, as the result of a design change, an elegant and formal statement of the requirements of a design step can be given as follows. Define:

- A specification S_1
- An implementation relation R
- A view function V , which has S_1 in its domain
- A view property P which is fulfilled by $V(S_1)$, i.e. $P(V(S_1))$ is true.
- A view property P' and a second view V' such that $\text{not}(P'(V'(S_1)))$

A **structured** design change corresponds to the specification of S_2 , the next design, such that:

- $R(S_1, S_2)$, and R is a strong bisimulation equivalence¹⁰.
- $P(V(S_2))$ and $P'(V'(S_2))$

In other words, S_2 maintains the external behaviour of S_1 , maintains the view property P and adheres to a new view property P' , which was not fulfilled by S_1 . One could say that the reason for defining S_2 was the fulfilment of this new property.

A **functional** design change corresponds to the specification of S_2 such that: $R(S_1, S_2)$, and R is an implementation relationship which is not a strong bisimulation equivalence. In other words, a functional design changes the behaviour tree of the specification being transformed. The effect of such a change on view properties is specific to each design.

Some design steps can be defined as a mixture of the structure and functional approaches. In such instances, the behaviour tree is changed *and* view properties are maintained. Design CPTs provide a means of generating a suitable S_2 from any given S_1 such that the appropriate properties and relations are guaranteed.

5.6.3.2 The CPT Formulation

Section 6.6.3.1 focuses on the notion of a relation between two already specified design stages. It is useful to express the CPT problem in terms of transformations and constraints. We wish to discover a transformation T such that:

Given any S_1 such that $P(V(S_1))$ and $\text{not}(P'(V'(S_1)))$, then:
 $R(S_1, T(S_1))$ and $P(V(T(S_1)))$ and $P'(V'(T(S_1)))$

A Simple CPT Example

A LOTOS specification of a system is as a set of communicating processes. At this stage of development, the specification (design) has no multiway synchronisation. Between each pair of communicating processes there may be more than one synchronisation gate. We want a transformation which creates a new specification which conforms to the no multiway synchronisation constraint whilst guaranteeing the new property that there must be at most one gate shared between processes. Further, we require that the new design is a valid implementation of the old design.

This can be more formally specified, using the above notation, as follows:

- V , the view function, is defined to return a set of (process identifier \times process identifier \times gate identifier) triples, such that:

$$(p_1, p_2, g) \in V(S_i) \Leftrightarrow$$

p_1 and p_2 are defined to synchronise on gate g in LOTOS specification S_i .

- P the internal property is defined on S_i as:

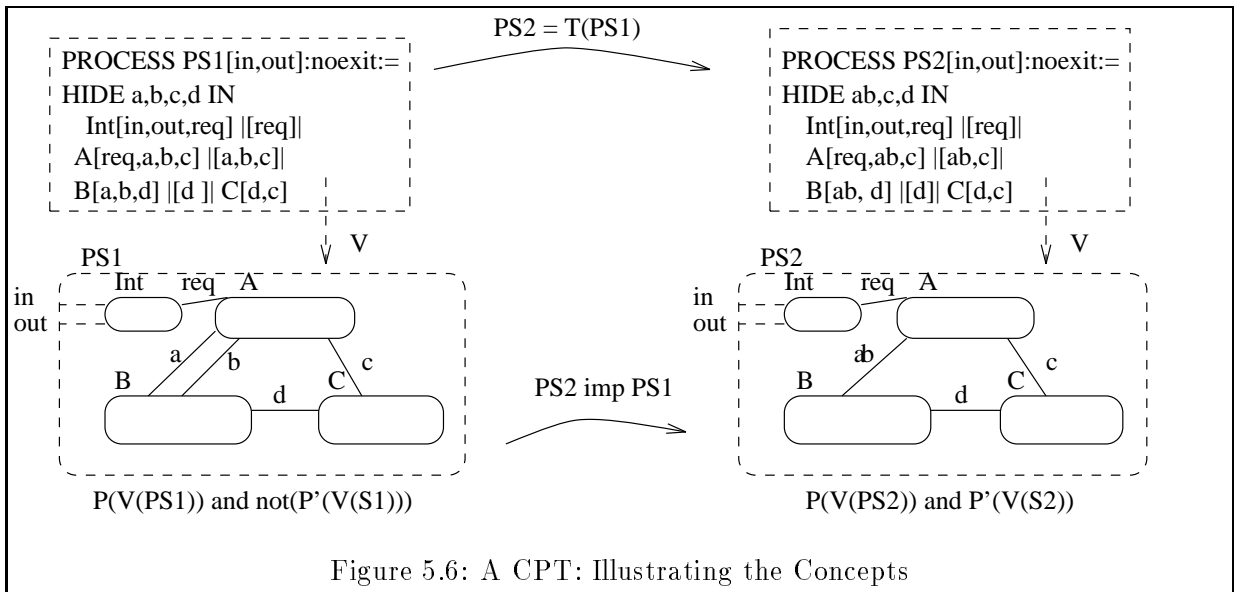
$$P(V(S_i)) \Leftrightarrow$$

$$((p, q, g), (r, s, g) \in V(S_i) \Rightarrow (((p = r) \Rightarrow (q = s)) \text{ or } ((p = s) \Rightarrow (q = r))))$$

¹⁰Strong bisimulation equivalence states that the behaviour trees offered by S_1 and S_2 are the same (even if the way in which they are specified is different).

- $V' = V$
- P' is defined on S_i as: $P'(V(S_i)) \Leftrightarrow ((p, q, g1), (r, s, g2) \in V(S_i) \Rightarrow (((p = r) \text{ and } (q = s)) \text{ or } ((p = s) \text{ and } (q = r)))) \Rightarrow g1 = g2)$

A transformation, T say, which fulfils these requirements is represented in diagrammatic form in figure 5.6.



5.6.4 Graphical Views and Tools

5.6.4.1 The Need For Graphics

Designs and structures are often represented quite naturally in graphical notation. It is therefore desirable to be able to **view** a system of parallel communicating LOTOS processes in such a way that it is possible to extract a unique, meaningful, graphical representation. Then, design decisions can be represented as transformations on a **view**, with all the advantages of an underlying formal method.

Chapters 2,3 and 4 introduce graphical views of static and dynamic behaviour as specified in OO ACT ONE. This thesis recognises that the presentation of graphical views of process algebra specifications is much more difficult (the language constructs are much more complex) than that for producing ADT views. A recent thesis by Winstanley [126] examines the graphical presentation of static and dynamic properties of process algebra specifications. This work, however, does not consider the presentation of object oriented properties. It is important that a graphical notation for our object oriented LOTOS designs emphasises object oriented aspects. Graphs are useful to help customers (and analysts) to understand requirements models. This thesis supports the opinion that similar views would be useful to help designers communicate with each other, and the programmers. The formality underlying the graphical models used during object oriented requirements capture must also be evident during design. Graphics should not be open to interpretation.

5.6.4.2 The Need For Tool Support

The LOTOS object oriented designs lend themselves to the production of some sort of automated formal design environment:

- There are a limited number of standard class and object representations, each with well-defined properties.
- The CPTs can be easily automated.
- There is potential for developing a graphical syntax for the representation of the OO designs, based on OO ACT ONE graphics and the communication model notation.
- There is potential for the execution of such designs using existing tools

Certainly, the designers can use the object oriented analysis tools and models to understand the underlying functional behaviour. However, separate tools are needed to help the analysis of the communication and interaction properties of the object oriented designs. Graphical tools are particularly important for the representation of structural properties.

5.6.5 CPT Driven Design: Some Other Concerns

5.6.5.1 Problems With The Dichotomy of LOTOS

It is much easier to reason about a system when there is a ‘conceptual consistency’ in the way it is specified. Conceptual consistency depends on a reasoned approach to the way in which a problem is decomposed into its component parts. In LOTOS, a behaviour can be specified with different emphasis placed on the roles of the data typing and process algebra. However, there has been little research into how this division takes place; and more particularly, why some specifiers favour one ‘half’ of the language over the other. A consistent specification approach requires that the roles of each ‘half’ of the language is clearly defined at each stage of the development. In practice, specifications do not seem to have this consistency. The object oriented development strategy in this thesis makes a clear distinction between the fundamental behaviour, as defined by the ACT ONE part of the specification, and the communication, timing and architectural aspects, as specified in the process algebra part.

A more pressing problem with full LOTOS, with respect to formalising transformations, is that proof systems for data algebras are generally distinct from proof systems for process algebras. Combining two systems in one coherent transformation proof framework is very difficult. This thesis avoids the problem of proving correctness in two different formal frameworks by maintaining the ADT part throughout the whole design process.

5.6.5.2 Practicality must be the driving force.

The notion of basing a whole development method on a CPT system is very tempting. However, we believe that, although the area of formal design is amenable to CPT techniques, it is not possible to force all design changes to be done using CPTs. CPT research must be driven by the needs of designers. At the moment, designers are repeatedly making the same sort of structure decisions on

different problems. These types of decisions must be identified and then formalised within the CPT framework: designers will then be able to incorporate formal techniques within their work without needing to directly manipulate the LOTOS code. There is potential for automation of the underlying formal transformations. This thesis gives only a flavour of what is possible. The CPTs are defined only in an object oriented framework and it is clear that many more CPTs are needed. CPTs must be the main tool for formal object oriented design.

5.6.6 Object Oriented LOTOS CPTs and the Resulting Design Trajectory

The CPTs in this thesis are defined only on LOTOS specifications which have been derived from the OO ACT ONE specification, using the initial transformation to LOTOS. A sequence of CPTs can be applied to this initial specification to result in a correctness preserving design trajectory. Within this trajectory CPTs can be applied to the specification components (and the components of the components ...). It is not necessary for all transformations being applied to be pre-defined CPTs. In some cases, a CPT may be identified which may be of use in many different problem domains, but is not yet formulated for re-use. It is recommended that, in such cases, the designer attempt to formulate such a general CPT (if they can). However, if this is not possible (or desirable) then the designers must verify the particular transformation which they employ. The formal object oriented design trajectory, which forms the basis of our object oriented development method, is illustrated in figure 5.7.

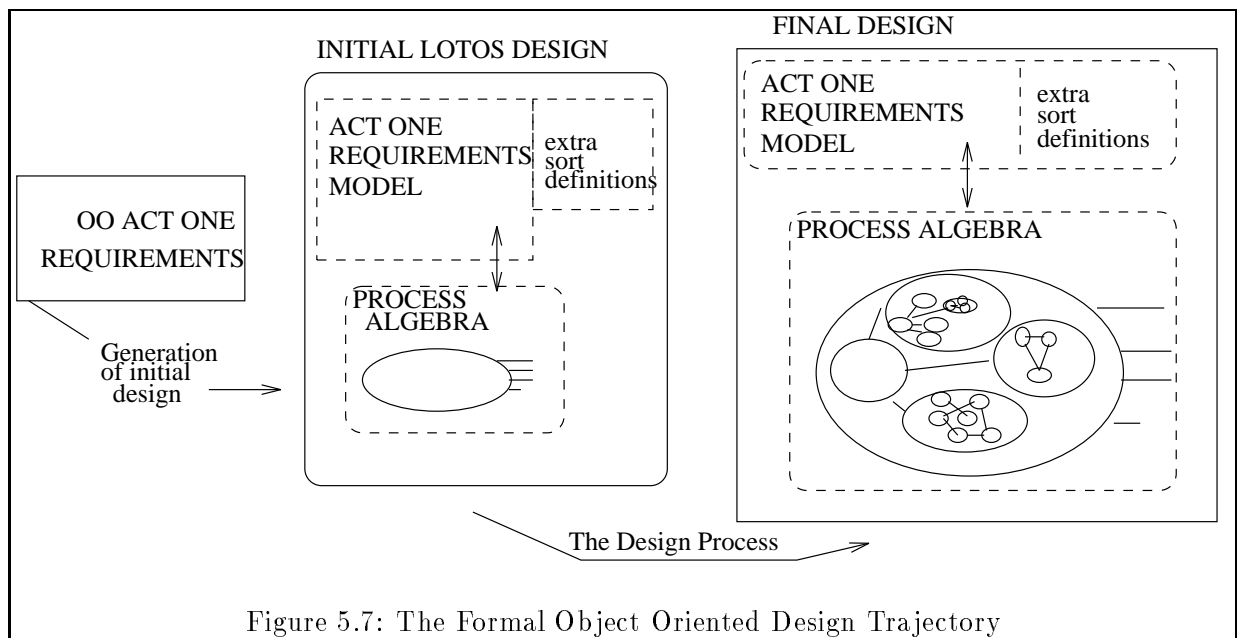


Figure 5.7: The Formal Object Oriented Design Trajectory

Notice that the ACT ONE code produced from the OO ACT ONE is maintained throughout development. There are two types of step in the design trajectory:

- A CPT-driven step, which does not need to be verified by the designer.

- A step, not achieved through application of a general CPT, which does need to be verified by the designer.

5.7 A Set of Object Oriented Design Decisions as CPTs

This section proposes five types of transformation which can be said to preserve the requirements as specified in the ACT ONE part of an initial LOTOS design. The transformations are used to illustrate the type of formal design which is possible within FOOD. The correctness of each transformation is argued informally: some rigorous reasoning is included, but it was beyond the scope of the thesis to prove the correctness of these transformations within the full LOTOS semantic framework. Future work must either:

- Define an object oriented design language whose semantics promotes the mathematical formulation of correctness and correctness preserving transformation.
- Address the problem of correctness formulation in full LOTOS, which arises out of the language being defined as a combination of an ADT and a process algebra. (The way in which our OO LOTOS specifications balance these parts of the specification makes this problem much more approachable than in the general case.)

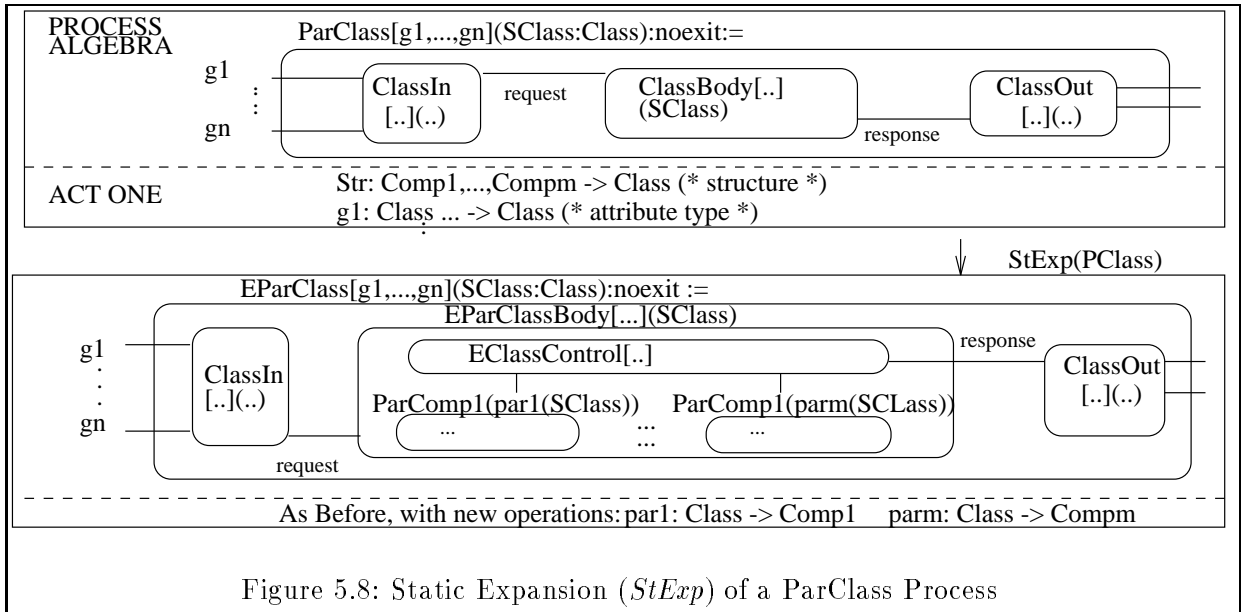
The five transformations which we define are as follows:

- *StExp* ('static expansion') is defined on the domain of Par and RCP process classes which have a fixed structure and are defined *purely*. This transformation replicates the structure of a class in the requirements model in the specification of a system of component processes.
- *Comp* ('composition') is defined on the domain of statically expanded process classes. It provides a means of re-grouping a subset (or subsets) of the components of a system.
- *Dist* ('distribution') is defined on the domain of Par processes which have been statically expanded. It provides a means of removing a centralised control by distributing the control amongst the component processes. It relies heavily on the multi-way synchronisation mechanism in LOTOS.
- *Rend* ('remove nondeterminism') is a simple mechanism for the removal of nondeterminism in the requirements model.
- Finally, a general technique (not identified by a particular transformation) for the removal of parallelism is proposed.

The case study, in chapter 7, requires the designs to be targetted towards an Eiffel implementation. Consequently, since Eiffel has a procedural model of communication, the case study does not illustrate the *Dist* transformation. However, it does illustrate: the static expansion of *purely* defined classes with fixed structure, composition as a means of restructuring and the removal of nondeterminism.

5.7.1 Static Structure Expansion

Expansion is the term given to any transformation which expands out the process algebra part of the OO LOTOS design with structure which is in the ADT part. The static expansion CPT (*StExp*) can be applied to Par and RPC class processes which have a fixed structure and *pure* specification. Informally, the static expansion transforms the unstructured class body process into a system of parallel processes. The static expansion of a **ParClass** process is shown in figure 5.8.



The static expansion of a **RPCClass** process is much simpler than that for the **ParClass**. It is illustrated in the design part of the case study (section 7.3). We do not report it here.

5.7.1.1 *StExp* Example: A System of Two Stacks

Consider a simple system of two stacks. The behaviour of the system is defined in the OO ACT ONE class **TwinStack**.

```

CLASS TwinStack USING Stack OPNS
STRUCTURES: TS< Stack, Stack >
TRANSFORMERS: push1<Nat>, push2<Nat>
DUALS: pop1 -> Nat, pop2 -> Nat
EQNS
TS(Stack1,Stack2).push1(Nat1) = TS(Stack1.push(Nat1), Stack2);
TS(Stack1,Stack2).push2(Nat1) = TS(Stack1, Stack2.push(Nat1));
TS(Stack1,Stack2).pop1 = TS(Stack1.pop, Stack2) AND Stack1..pop;
TS(Stack1,Stack2).pop2 = TS(Stack1, Stack2.pop) AND Stack2..pop
ENDCLASS (* TwinStack *)

```

The initial LOTOS design for this behaviour is generated using the *MakePar* mapping. This design is specified in process **ParTwinStack**. The process algebra specification for **PTwinStack** is given below.

```

process ParTwinStack[push1, push2, pop1, pop2 ](STwinStack):noexit:=
hide request, response in
TwinStackIn[ push1,push2,pop1,pop2,request ](0) |[ request ]|
TwinStackBody [ request, response ](STwinStack) |[ response ]|
TwinStackOut[ pop1, pop2, response ](0) where ...

```

The *StExp* CPT takes the `PTwinStack` process definition and produces a new process definition, named `EParTwinStack`. The CPT does this by leaving the definitions of the `TwinStackIn` and `TwinStackOut` processes alone whilst changing the `TwinStackBody` process specification. This is specified below.

```

process EParTwinStack[push1, push2, pop1, pop2 ](STwinStack):noexit:=
hide request, response in
TwinStackIn[ push1,push2,pop1,pop2,request ](0) |[ request ]|
EParTwinStackBody [ request, response ](STwinStack) |[ response ]|
TwinStackOut[ pop1, pop2, response ](0)
where ...

```

The new `EParTwinStackBody` is defined as a structured process in which there are three component (sub)processes:

- A control process, named `EParTwinStackControl` by convention, which, as its name suggests, controls the way in which the other components interact to produce the required behaviour.
- Two `ParStack` component processes: one for each component of the **structure** operation `TS`.

```

process EParTwinStackBody[request, response](STwinStack):noexit:=
hide Stack1push, Stack1pop, Stack2push, Stack2pop in
EParTwinStackControl[ Stack1push, Stack1pop, Stack2push, Stack2pop, request, response ](0)
|[ Stack1push, Stack1pop, Stack2push, Stack2pop ]|
( ParStack [ Stack1push, Stack1pop ](par1(STwinStack)) |||
ParStack [ Stack2push, Stack2pop ](par2(STwinStack)) )
where ...

```

There are a number of things worth noting about this specification, before details of the `EParTwinStack` process are considered.

- The hidden gates, namely `Stack1push`, `Stack1pop`, `Stack2push`, `Stack2pop`, have a 1-1 correspondence with the set of external attributes offered by the component classes of the `TwinStack`. These gates are identified by the component class name, followed by the parameter index of that class in the structure operation and finished by the attribute name.
- New sort operations, namely `par1` and `par2`, are used to return the individual parameter values of any given `TwinStack` `TS` structure representation. These new operations are generated by the *StExp* transformation, and added to the ADT part of the specification.
- The composition structure of the `TwinStack` has been expanded out in the process algebra part of the resulting design. This structure is still present in the ADT part, but it is now explicit in the communications model.

- The underlying functionality is contained in the `ParStack` components.

The *StExp* transformation is concerned mainly with the generation of a suitable `Control` process for any given statically structured class. `ETwinStackControl` illustrates how such a process is generated, for a simple behaviour.

```

process EParTwinStackControl[ Stack1push, Stack1pop, Stack2push, Stack2pop, request, response ]:
noexit:=
(request!push1?Nat1:Nat?ID:Nat;
Stack1push!Nat1; (EParTwinStackControl[...] ||| (response!push1!ID; exit))[]
(request!pop1?ID:Nat;
Stack1pop; Stack1pop?Result:Nat;
(EParTwinStackControl[...] ||| (response!pop1!Result!ID; exit))[]
(request!push2?Nat1:Nat?ID:Nat;
Stack2push!Nat1; (EParTwinStackControl[...] ||| (response!push2!ID; exit))[]
(request!pop2?ID:Nat;
Stack2pop; Stack2pop?Result:Nat;
(EParTwinStackControl[...] ||| (response!pop2!Result!ID; exit))
endproc (* EParTwinStackControl *)

```

The `EParTwinStackControl` specification is simple to generate because the two `Stack` components are not configured. The external attributes of the `TwinStack` are serviced by the `Control` ‘passing them on’ to the components, using the new internal gates.

Consider now extending the `TwinStack` behaviour with a `swaptops` transformer such that `Stack1` and `Stack2` are configured on `swaptops`. The `swaptops` attribute is more formally defined as:
`TS(Stack1,Stack2).swaptops = TS((Stack1.pop).push(Stack2..pop), (Stack2.pop).push(Stack1..pop));`
This attribute is translated into the `EParTwinStackControl` process by the inclusion of a new choice behaviour expression:

```

...[] (request!swaptops?ID:Nat;
(( Stack1pop; Stack1pop?Result1:Nat; exit)
|||
( Stack2pop; Stack2pop?Result2:Nat; exit ) )>>
(( Stack1push!Result2; exit) ||| ( Stack2push!Result1; exit) )>>
(EParTwinStackControl[...] ||| (response!swaptop!ID; exit)) ) ...

```

This more complex attribute gives a better flavour of how, in general, external attributes are translated by the *StExp* CPT. The resulting behaviour expression is made up of four parts:

- i) Accept the request and input parameters.
- ii) Perform internal accessor and dual operations for each dependent component.
- iii) Use the information gathered, if necessary, to perform internal state transitions (via external transformer requests).
- iv) Offer the response (with result in the case of a dual or accessor) in parallel with the original `Control` behaviour

When an external attribute depends on only one component then the translation is simplified by not having to use the $|||$ or $>>$ operators. The `push1`, `pop1`, `push2` and `pop2` attributes, in `TwinStack`, are a good example of this.

5.7.1.2 The Correctness Of The Static Expansion

We are required to prove that `ETwinStack` is a **class implementation**¹¹ of `PTwinStack`. In other words, given `STwinStack`, a value of the `TwinStack` sort, then `ETwinStack[...](STwinStack) impl PTwinStack[...](STwinStack)`. The *StExp* relation between these two behaviours is illustrated in figure 5.9.

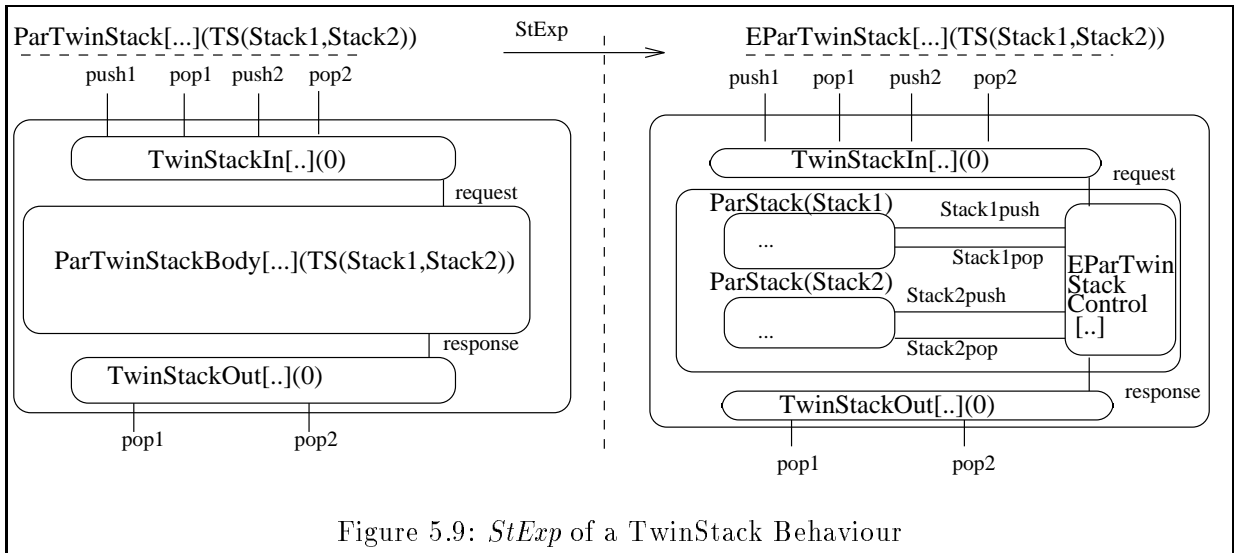


Figure 5.9: *StExp* of a `TwinStack` Behaviour

The two specifications have `TwinStackIn` and `TwinStackOut` components in common. The only difference is between the unstructured `ParTwinStackBody` and the structured `EParTwinStackBody` process instances. The `EParTwinStack` is an implementation of the corresponding `ParTwinStack` instance because of a simple property of the **implementation** relationship: any behaviour expression which contains an internal event can have that internal event transformed into a sequence of internal events (or vice-versa) without changing the external behaviour of the expression. The *StExp* transformation changes internal requests (and responses) into sequences of internal events which model the passing on of the requests to the component processes, and responses back again.

The `TwinStackIn` and `TwinStackOut` processes, common to the design specification before and after the transformation, guarantee the external ordering of service requests and responses, no matter what changes are made to the internal sequence of events. Further, the use of the ADT specification to provide the underlying functionality guarantees compatibility between the behaviours offered. The transformation cannot introduce livelock or deadlock and so correctness is preserved.

¹¹The **implementation** relationship is one which guarantees the preserving of the requirements in the original OO ACT ONE model.

5.7.1.3 Other Complexities To Be Addressed

The `TwinStack` example was chosen for its simplicity, and as such it does not address all the complexities of the transformation. These are as follows:

- **Preconditioned Equations in the Requirements Model.**

Preconditioned equations are defined on structured classes as boolean expressions which depend on accessor¹² service replies from components. Preconditioned equations translate quite naturally into guarded expressions in the process algebra. Note that the completeness of the preconditioned equation (guaranteed by the `OTHERWISE` construct) means that no deadlocks can be introduced by the generation of guarded expressions in the new design specification. Further, no additional nondeterminism can arise from more than one guarded expression being true (across a choice of behaviours).

- **Invariants in the Requirements Model.**

Invariants are realised by ‘global preconditions’ on every attribute in a class. Consequently, invariants are translated into ‘global guards’ in the process algebra. This can, unfortunately, lead to deadlocks when invariants are not proven, in the analysis stage, to be maintained throughout the lifetime of an object.

The complexities arising from the handling of preconditions and invariants are not considered in any of the CPTs that follow.

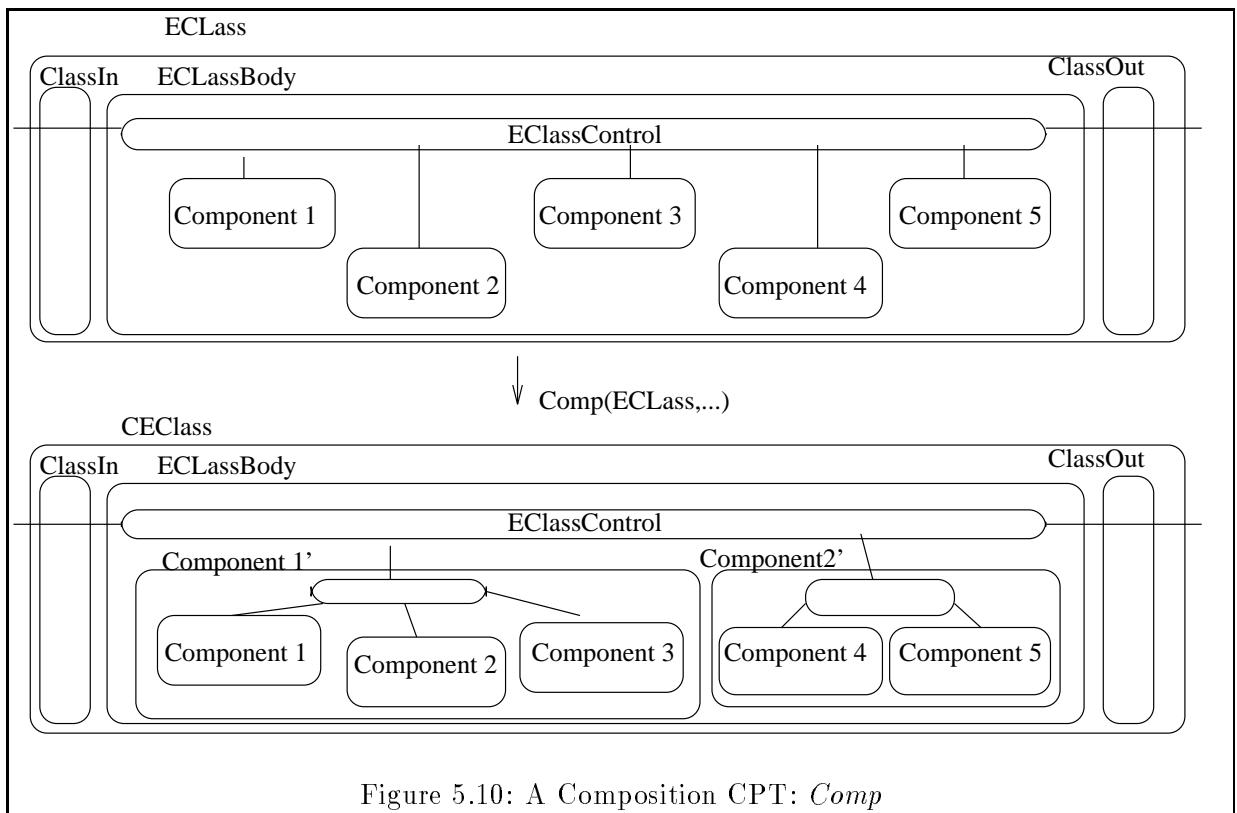
5.7.2 Compositional Re-Structuring For Re-Use

There are two important aspects to restructuring for re-use: decomposition and composition. It is necessary to be able to decompose larger components into smaller ones so that the smaller components which have already existing implementations can be re-used. The static expansion transformation (*StExp*) provides a decomposition mechanism. It is also necessary to be able to compose smaller components into larger ones so that the implementation of the larger component can be re-used. It is this type of transformation which is considered in this section.

A simple solution to the re-structuring problem is to define a CPT which is the inverse of the *StExp* CPT. However, this is not general enough, since the designer may wish to combine only a subset of the component parts rather than all of them. Consider a `Class` which has five components. The designers wish to combine components 1,2 and 3, and components 4 and 5 to create new components (component1’ and component2’). These new design components correspond to some already implemented behaviour which can be re-used directly. This restructuring is illustrated in figure 5.10.

The *Comp* CPT acts on any given statically expanded LOTOS specification. It is parameterised on a partitioning of the component set. In the diagram above, the partitioning is: $\{\{component1, component2, component3\}, \{component4, component5\}\}$.

¹²In the requirements model preconditioned equations can be defined only on accessor attributes so that component state changes cannot arise from the evaluation of the precondition boolean expression.

Figure 5.10: A Composition CPT: *Comp*

5.7.2.1 *Comp* CPT Example

Consider a system of two queues (of natural numbers) and a transformer. The system accepts **Nats** via the **on** attribute, transforms and then queues them up via the internal **trans** attribute, and outputs the results in their original order via the **off** attribute. This is more precisely specified by the OO ACT ONE **System** class definition, below.

```

CLASS System USING Queue, Transformer OPNS
STRUCTURES: SQQT < Queue, Queue, Transformer >
TRANSFORMERS: on<Nat>, trans (* internal *)
DUALS: off -> Nat
EQNS
SQQT(Queue1,Queue2,Transformer1).on(Nat1) = SQQT(Queue1.push(Nat1),Queue2,Transformer1);
SQQT(Queue1, Queue2, Transformer1).trans =
  SQQT(Queue1.pop, Queue2.push(Transformer1.tr(Queue1..pop)), Transformer1);
SQQT(Queue1,Queue2,Transformer1).off = SQQT(Queue1,Queue2..pop,Transformer1) AND Queue2..pop
ENDCLASS (* System*)

```

The **System** class structure diagram is represented in figure 5.11.

Static expansion of the **ParSystem** process results in the **EParSystem** process definition, as partially defined by the **EParSystemBody** process, below (the other parts of the **EParSystem** specification are not affected by the *Comp* transformation).

Now, the designers may be aware of a precoded component, **DoubleQ** say, which provides the

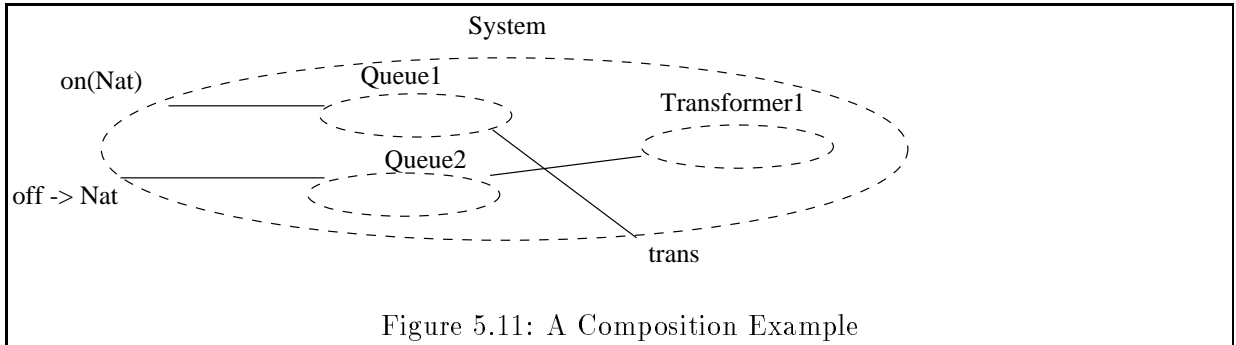


Figure 5.11: A Composition Example

```

process EParSystemBody[request, response] (SSystem:System): noexit :=
hide Queue1push, Queue1pop, Queue2push, Queue2pop, Transformer1tr in
SystemControl[...]|[...]
( ParQueue[ Queue1push, Queue1pop] (par1(SSystem)) |||
ParQueue[ Queue2push, Queue2pop] (par2(SSystem)) |||
ParTransformer[ Transformer1tr] (par3(SSystem)) )
where ...
endproc (* ESystemBody *)

```

functionality of two natural number queues which are linked in some unspecified way. Rather than having two distinct `Queue` components in the design, it is advantageous to combine them together into a single component. This can be done using the *Comp* CPT.

$Comp(ESystem, \{\{1,2\}, \{3\}\})$ results in a new process specification which differs from the old process only in the specification of the `ClassBody`. The new process class body is named `CEClassBody`, in this case. The specification of `CESystemBody`, resulting from $Comp(ESystem, \{\{1,2\}, \{3\}\})$ is given below.

```

process CESystemBody[request, response] (SSystem:System): noexit :=
hide Queue1push, Queue1pop, Queue2push, Queue2pop, Transformer1tr in
SystemControl[...]
|[...]|(
CQueueQueue[Queue1push,Queue1pop,Queue2push,Queue2pop] (par1(SSystem), par2(SSystem))
||| ParTransformer[ Transformer1tr] (par3(SSystem))
)where
process CQueueQueue[Queue1push,Queue1pop,Queue2push,Queue2pop]
(Queue1:Queue,Queue2:Queue):noexit:=
ParQueue[ Queue1push, Queue1pop] (Queue1) ||| ParQueue[ Queue2push, Queue2pop] (Queue2)
endproc (* CQueueQueue *) ...endproc (* ESystemBody *)

```

The new `CQueueQueue` process can now be implemented using the pre-coded `DoubleQueue` component. For consistency, it is beneficial to be able to respecify the `CQueueQueue` process in standard `ParClass` form. Then, it can be transformed by any of the design CPTs. This standardisation requires the creation of a new ADT class, defined as a static structure with two `Queue` components. In other words, the ADT model of the new component is reverse engineered into a new OO ACT ONE class specification.

5.7.2.2 An Overview of the Correctness of *Comp*

The *Comp* CPT is doing nothing more than bracketing together sequences of interleaved operations and substituting internal events with sequences of internal events. Since the $|||$ operator is associative and the components being combined are interleaved processes, any bracketing of these processes can be done without altering the behaviour being specified.

5.7.2.3 Limitations of *Comp*

The *Comp* CPT works only on LOTOS specifications which have expanded static structure, with a centralised control component. Part of the job of an object oriented designer is to distribute the control aspects of a system among its component parts (see the *Dist* CPT in section 5.7.3). This distribution often means that the component processes are no longer interleaved, but must synchronise on shared gates. It is much more difficult to formulate a composition CPT for these types of distributed system specifications. This line of research is not examined in the design part of this thesis. Rather, during design we recommend that the expanded class specifications are compositionally restructured before the *Dist* CPT is applied.

5.7.3 Re-Structuring for Distributed Control

All structured LOTOS *EParClass* processes have a centralised control to manage the way in which the component processes are used to provide the external functionality. The *EParClass* processes have a structure as shown in the left hand side of figure 5.12. The *Dist* CPT produces a *DistClass* structure, as shown on the right hand side of the same figure.

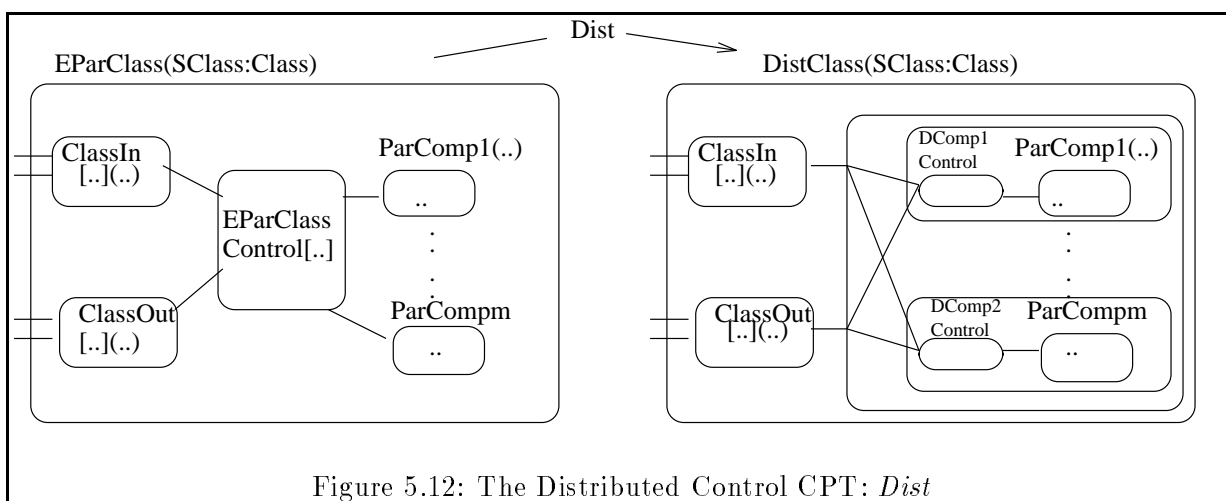


Figure 5.12: The Distributed Control CPT: *Dist*

Before the *Dist* CPT is applied, the *ClassBody* process has its concurrent processes under the control of the *ClassControl* process. In effect, there is a centralised process through which all requests and responses go. Object oriented designers may wish to remove this centralisation and distribute control in a decentralised fashion. There are potentially an infinite number of ways in which a designer could choose to do this. This section defines one CPT, namely *Dist*, which distributes the centralised

control of a `EClass` process amongst all of the `Class` component processes. The *Dist* CPT is defined on `EParClass` processes.

5.7.3.1 *Dist* of NonConfigured Structure: a TwinStack Example

Consider the `TwinStack` behaviour defined in 5.7.1. The *Dist* CPT applied to `EParTwinStack` produces the `DistTwinStack` process specification, below.

```

process DTwinStack[push1, push2, pop1, pop2 ](STwinStack):noexit:=
hide request, response in
TwinStackIn[ push1,push2,pop1,pop2,request ](0) |[ request ]|
DTwinStackBody [ request, response ](STwinStack: TwinStack) |[ response ]|
TwinStackOut[ pop1, pop2, response ](0)
where
(* TwinStackIn and TwinStackOut are specified as before *)
process DTwinStackBody[request, response](STwinStack: TwinStack):noexit:=
DStack1[request,response](par1(STwinStack))
|||
DStack2[request,response](par2(STwinStack))
where ...
endproc (* DTwinStackBody *)
endproc (* DTwinStack *)

```

In the `EParTwinStack` class, the `ParStack` components are not configured. Consequently, there is no need for synchronisation between the `DStack1` and `DStack2` components of `DistTwinStackBody`. These two processes are interleaved to provide the required behaviour. Their specifications are given below.

```

process DStack1[request,response](SStack:Stack):noexit:=
hide Stack1push, Stack1pop in
ParStack[ Stack1push, Stack1pop ](SStack) |[ Stack1push, Stack1pop ]|
DStack1Control[ request, response, Stack1push, Stack1pop ]
where
(* ParStack is specified in the normal way *)
process DStack1Control[ request, response, Stack1push, Stack1pop ] :noexit:=
(request!push1?Nat1:Nat?ID:Nat; Stack1push!Nat1;
(DStack1Control[...] ||| response!push1!ID; exit)) []
(request!pop1?Nat1:Nat?ID:Nat; Stack1pop; Stack1pop?Result:Nat;
(DStack1Control[...] ||| response!pop!Result!ID; exit))
endproc (* DStack1Control *) endproc (* DStack1 *)
process DStack2 ... (* Defined similarly to DStack1 *)

```

5.7.3.2 Distribution of Configured Structure: A TwinStack Extension Example

Consider an extension of the `TwinStack` behaviour in which the two components are configured by a `swaptops` attribute, which is defined as:

```

TS(Stack1,Stack2).swaptops =
  TS((Stack1.pop).push(Stack2..pop), (Stack2.pop).push(Stack1..pop));

```

This results in the extended `TwinStackControl` process as defined earlier in this section. We must consider how such a configuration is transformed by the *Dist* CPT. Intuitively, there now must be some sort of internal gate shared by the component `Stacks`. The passing of information between components, which was originally done by the centralised control process, must now be done by the components.

In the *Dist* transformation, when processes i_1, \dots, i_r are configured by an external attribute, then an internal configuration gate named `config $i_1 \dots i_r$` is defined in the resulting `ClassBody`. For example, since components 1 and 2 configure in the `TwinStack` (on the `swaptops` attribute) there is an internal gate defined as `config12` in the `DistTwinStackBody` process.

In the case where class components are configured, the *Dist* CPT produces a structured control process for every component. Each control is made up of `ServiceThese` and `IgnoreThese` components running in parallel, but not synchronised. The `ServiceThese` process controls the servicing of requests which depend on the particular component which it is controlling. The `IgnoreThese` process participates in all requests and responses which do not depend on the component to be fulfilled. This is necessary because *all* components must multi-way synchronise on request and response events. The `PStack` process, running in parallel with the control process, is defined in the normal way. Consequently, it can also be manipulated using CPTs whilst maintaining correctness. This is illustrated in the following code for process `DistTwinStack`.

The `DistTwinStack` example illustrates quite clearly how the configured servicing of a transformer `swaptops` is distributed amongst the two components. It is not clear, without further investigation, whether the distribution of control is as straightforward for dual (and accessor) attributes. In a ‘well defined’ OO ACT ONE specification of requirements, the result of a dual (and accessor) is always the result of a dual (or accessor) at one of the components of the structure. It is therefore quite natural in the distributed design for this one component to take responsibility for the result response. For example, consider a dual attribute `op` defined on a `TwinStack` as follows:

```

TS(Stack1, Stack2).op = TS((Stack1.pop).pop, Stack2.push(Stack1..pop)) AND T(Stack1.pop)..pop;

```

This results in additional fragments of LOTOS code in the distributed `TwinStack` design: the two `Control` processes, in each `Stack` component, are extended in the code below.

5.7.3.3 Overview of the *Dist* CPT Definition

The main complexity in the definition of *Dist* is the analysis of the configured attribute requirements. This analysis must identify whether attributes configure components. The parsing of the configured attribute requirements then splits the service into four parts:

- Performs accessors (and duals) on components which provide results for use in the internal requests in the remainder of the service. In the transformation, these result in a set of parallel internal events with ‘data flow’ modelled by the internal `config` event synchronisations.
- Dual events must be then be processed in order of nesting The ordering is maintained by the control parts of each component.

```

process DistTwinStackBody[request,response] (STwinStack:TwinStack):noexit:=
hide config12 in
DStack1[ request, response, config12](par1(STwinStack))
|[ request, response, config12 ]|
DStack2[ request, response, config12 ](par2(STwinStack)) where
process DStack1[ request, response, config12 ] (SStack: Stack):noexit:=
hide Stack1push, Stack1pop in
PStack[ Stack1push, Stack1pop ](SStack) |[ Stack1push, Stack1pop ]|
DStack1Control [ request, response, Stack1push, Stack1pop ] where
(* PStack is defined in the normal way *)
process DStack1Control[...]:noexit:=
ServiceThese[request, response, Stack1push, Stack1pop] |||
IgnoreThese[ request, response ] where
process ServiceThese[ request, response, Stack1push, Stack1pop ]:noexit:=
(request!push1?Nat1:Nat?ID:Nat; Stack1push!Nat1; (ServiceThese[...] |||
response!push1!ID; exit)) []
(request!pop1?Nat1:Nat?ID:Nat; Stack1pop; Stack1pop?Result:Nat; (ServiceThese[...] |||
response!pop!Result!ID; exit))
(request!swaptops?ID:Nat; Stack1pop; Stack1pop?Result1:Nat; config12!Result1?Result2:Nat;
Stack1push!Result2; (ServiceThese[...] |||
response!swaptops!ID; exit))
endproc (* ServiceThese *)
process IgnoreThese[ request, response ]:noexit:=
(request!push2?Nat1:Nat?ID:Nat; IgnoreThese[...]) []
(request!pop2?Nat1:Nat?ID:Nat; IgnoreThese[...]) []
(response!push2!ID:Nat; IgnoreThese[...]) []
(response!pop2?Result:Nat?ID:Nat; IgnoreThese[...])
endproc (* IgnoreThese *) endproc (* DStack1Control*) endproc (* DStack1*)
process DStack2[ request, response, config12 ] (SStack: Stack):noexit:=
(* Defined similarly to DStack1 *)

```

- The additional internal services that are required to achieve the correct global state of the system are treated separately as the penultimate part of the distributed service.
- Finally, the analysis identifies the component which is responsible for returning the result of the request (if it has a result). The response event is synchronised on by all components, but only one provides the result (the others accept any result value).

5.7.3.4 Overview of the Correctness of *Dist* On Configured Expanded Classes

As for the other structural CPTs, *Dist* does not change the external functionality of the system (class) being specified: it restructures the internal events (or sequences of events) which control the interaction between components of the system. Rather than having one central control process, the control is distributed amongst the components using multi-way synchronisation. Each component then decides which service requests it has to be involved in. Correctness is guaranteed because the *ServiceThese* and *IgnoreThese* processes guarantee the non-introduction of internal deadlock or livelock, the *TwinStackIn* and *TwinStackOut* processes maintain the same external communication

```

process DStack1Control[...]:noexit:=
ServiceThese[request, response, Stack1push, Stack1pop] |||
IgnoreThese[ request, response ] where
process ServiceThese[ request, response, Stack1push, Stack1pop ]:noexit:= ...
(request!op?ID:Nat; Stack1pop; Stack1pop?Result1:Nat; config12!Result1; Stack1pop?Result2:Nat;
(ServiceThese[...] |||
response!op!Result2!ID; exit)) ...
endproc (* ServiceThese *)
(* Process IgnoreThese defined as before *)
endproc (* DStack1Control*)

process DStack2Control[...]:noexit:=
ServiceThese[request, response, Stack1push, Stack1pop] |||
IgnoreThese[ request, response ] where
process ServiceThese[ request, response, Stack1push, Stack1pop ]:noexit:= ...
(request!op?ID:Nat; config12?Result1:Nat; Stack2push?Result1:Nat;
(ServiceThese[...] ||| response!op?Result2:Nat?ID:Nat; exit)) ...
endproc (* ServiceThese *)
(* Process IgnoreThese defined as before *)
endproc (* DStack2Control*)

```

interface, and the ACT ONE part of the design maintains the external functionality.

5.7.3.5 The Importance of the Distribution CPT *Dist*

The *Dist* CPT is the first step towards the formalisation of very complex systems of distributed objects (processes). It introduces the possibility of modelling concurrent objects and shared objects at the high levels of design. This thesis is not concerned with the development of distributed software. However, the *Dist* CPT does illustrate how such work may be instigated in FOOD. There is much scope for developing a set of CPTs which can be applied to distributed `DistClass` processes.

5.7.4 Resolving Explicit NonDeterminism

This section addresses the need for designers to remove nondeterminism in specifications. The CPT which we examine in this section is concerned with removing the nondeterminism due to `(* INTERNAL *)` transformations in the requirements model. One approach to removing nondeterminism is provided by the *Rend* ('remove nondeterminism') CPT.

5.7.4.1 Resolving Explicit NonDeterminism Using *Rend*: A CoinToss Example

Reconsider the simple `CoinToss` Class in section 4.3.4. The O-LTSD is given, in figure 5.13, as a reminder of its behaviour.

The LOTOS `ParCoinToss` process, defined below, is the first high-level object oriented design of this behaviour¹³.

¹³The removal of nondeterminism in the other types of object oriented LOTOS specifications is done similarly.

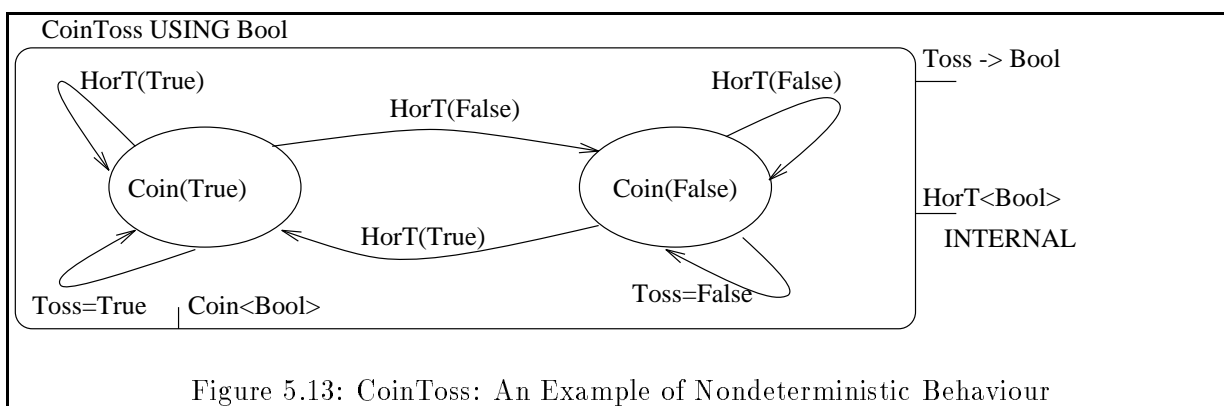


Figure 5.13: CoinToss: An Example of Nondeterministic Behaviour

```

process PCoinToss[ Toss ] (SCoinToss: CoinToss):noexit:=
hide request, response, HorT in
CoinTossIn[ Toss, request, HorT ](0) |[ request ] |
CoinTossBody[ request, response ](SCoinToss) |[ response ] |
CoinTossOut[ Toss, response ](0) where ...
(* These processes are defined as if HorT was an external attribute. *)

```

The **PCoinToss** specification says nothing about how, why or when the **HorT** internal transitions occur. Clearly, the designers must resolve this nondeterminism before implementation can begin. The *Rend* approach requires the specification of a new process which runs in parallel with the **CoinTossIn** process. This new process restricts when the internal transitions can take place. The *Rend* CPT takes a process, **DetCoinToss** say, as a parameter and produces a new **DetPCoinToss** specification, as defined below.

```

process DetPCoinToss[ Toss ] (SCoinToss: CoinToss):noexit:=
hide request, response, HorT in
( DetCoinToss[ request, HorT ] |[ request, HorT ] |
CoinTossIn[ Toss, request, HorT ](0) )
|[ request ] |
CoinTossBody[ request, response ](SCoinToss) |[ response ] |
CoinTossOut[ Toss, response ](0) where ...

```

DetCoinToss can be any process specification which has a gate list **[request, HorT]** and is of type **noexit**. The correctness of the *Rend* transformation on **PCoinToss** depends on **DetCoinToss** fulfilling a simple property: at any stage in the behaviour of **DetCoinToss**, all external attribute request events (of the correct form) must be offered immediately or after a finite number of **HorT** events. This property guarantees the correctness of the *Rend* transformation. *Rend* places the responsibility on the designers to prove that the required property is upheld. Fortunately, as the examples below show, this is often trivial.

5.7.4.2 A Set of More Deterministic Coin Tosses

In this section we define a set of `DetCoinToss` processes, each of which models a different way of removing some, or all, of the *HorT* nondeterminism in the `CoinToss` behaviour.

- I)

```
DetCoinToss[...]: noexit :=
  request!Toss; HorT?Bool1:Bool; request!HorT!Bool1?Nat1:Nat; DetCoinToss[...]
```

In this specification, `DetCoinToss` resolves only some of the nondeterminism by stating that after every `request!Toss` a state transition must take place before another `request!Toss` can be accepted. It says nothing about what state transitions occur between tosses.

- II)

```
DetCoinToss[...]: noexit :=
  HorT!true; request!HorT!true?Nat1:Nat; request!Toss;
  HorT!false; request!HorT!false?Nat2:Nat; DetCoinToss[...]
```

In this case, the designers resolve the nondeterminism by stating that the coin tosses true and false, alternately.

- III)

```
DetCoinToss[...]: noexit :=
  HorT!true; request!HorT!true?Nat1:Nat; request!Toss; DetCoinToss[...]
```

In this case, the designers resolve the nondeterminism by stating that the coin always tosses true.

These simple examples show the power in separating out the explicit resolution of internal transitions from the rest of the `DetClass` behaviour. The *Rend* CPT shows only one mechanism of resolving explicit nondeterminism in a controlled fashion. The domain of the *Rend* transformation is `ParClass` process specifications, but this can be easily extended. Like all the CPTs put forward in this thesis, *Rend* is used to show only that CPT based-design, in an object oriented LOTOS framework, has the potential for practical application.

5.7.5 Removing Parallelism

The object oriented LOTOS specifications, in this work, model concurrency using the parallel operators `|||`, `||` and `| [...] |`. Two processes combined by the parallel operator(s) can be said to be concurrent — of course the concurrency is just represented by an arbitrary interleaving of events. If the target implementation language supports concurrent entities then it is the job of the designers to match LOTOS processes to these entities. However, designers may wish to remove the parallelism when it is not supported at the implementation level, or if it is too fine-grain to warrant a mapping to separate implementation entities.

5.7.5.1 Removing Arbitrary Interleaving In Behaviour Expressions

The extension CPTs tend to produce design specifications in which parallelism models the arbitrary interleaving of communication events between a centralised control process and the component processes of which it requests services. For example, the `ETwinStack` services the `swaptops` attribute in the following way:

```
request!swaptops?ID:Nat;
((Stack2pop; Stack2pop?Result1:Nat; exit) ||| (Stack1pop; Stack1pop?Result2:Nat; exit))>> ...
```

The order in which the elements are popped off the two `Stack` components is not determined by the `ETwinStack` design. This leaves the designers some implementation freedom: the `TwinStack` may access the information concurrently or it may do it sequentially. The designer is free to remove the parallelism by changing the attribute definition. For example, a design decision to access `Stack1`, followed by `Stack2`, results in the following code:

```
request!swaptops?ID:Nat;
(Stack1pop; Stack1pop?Result1:Nat; Stack2pop; Stack2pop?Result2:Nat;)>> ...
```

Rather than attempting to specify a CPT which controls this type of design decision, we say that any behaviour expression in the object oriented designs written as $(P; \text{exit}) \parallel (Q; \text{exit})$ can be transformed into $(P; Q; \text{exit})$ or $(Q; P; \text{exit})$ whilst preserving correctness.