

Chapter 1

Introduction

Formal languages, based on mathematical models, are essential in achieving *correct* software. *Correctness*, being a mathematical proof that software fulfils requirements, depends on: a formal requirements model, a formal implementation model and a means of relating the two. These requirements appear straightforward until the complex nature of software is considered.

Software development has reached the point where the complexity of the systems being modelled cannot be handled without a thorough understanding of underlying fundamental principles. Such understanding forms the basis of scientific theory as a rationale for software development techniques which are successful in practice. This scientific theory, as expressed in rigorous mathematical formalisms, must be transferred to the software development environment. Only then can the development of software systems be truly called *software engineering*: the application of techniques, based on mathematical theory, towards the construction of abstract machines as a means of solving well defined problems.

Object oriented methods encompass a set of techniques which have been, and will continue to be, applied in the successful production of complex software systems. The methods are based on the simple mathematical models of abstraction and classification. Further, the conceptual consistency offered by the object oriented paradigm across the software development process, together with an emphasis on re-use, promotes very fast code production cycles. However, the inherent flexibility in object oriented development environments often leads to an interactive and incremental style of development, a consequence of which may be insufficient rigour in software production. This lack of rigour is re-enforced by the informal set of object oriented concepts and terminology which, although applied in all stages of development and based on the same underlying principles, mean different things to different people.

Object oriented methods can be used to aid the construction of formal models. Formality can help to improve object oriented software development techniques. Object oriented and formal methods are complementary: their integration has the potential for producing a software development environment which incorporates the benefits of both. Fundamental to the successful integration of mathematical rigour and object oriented methodology is a formal model of object oriented concepts which is consistent throughout the whole software development process. Such a model forms the basis

of the work put forward by this thesis.

1.1 Scope: Software Engineering

Software engineering, formal methods and the object oriented paradigm together form an enormous body of work, much of which is beyond the scope of this thesis. The software engineering boundaries of the work are clearly defined in figure 1.1. The analysis, requirements capture, design and implementation models and methods put forward by this thesis are collectively named FOOD: *Formal Object Oriented Development*.

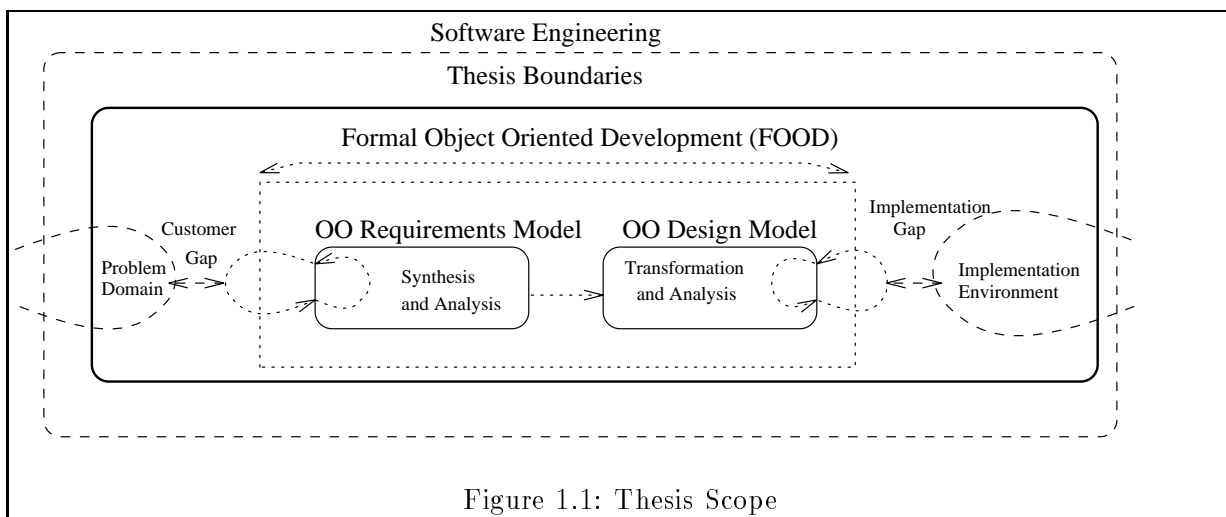


Figure 1.1: Thesis Scope

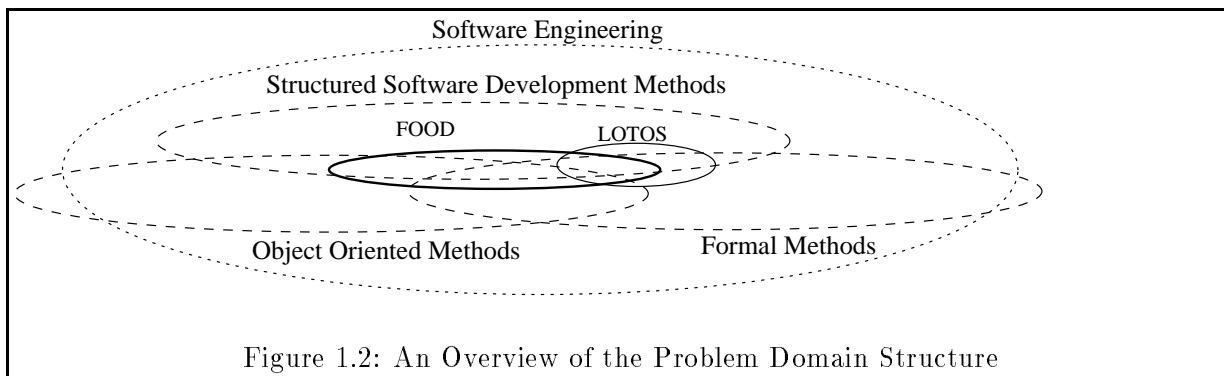
FOOD is principally concerned with maintaining correctness between the initial *customer oriented* requirements model and the final *implementation oriented* design. The formal boundaries break down at either end of the software development process because, in general, target implementation languages are not formally defined and customer understanding of their requirements is not complete. An object oriented approach helps to bridge these two gaps because of the conceptual consistency.

The thesis makes a clear distinction between models and methods: models provide only a framework upon which software can be constructed, whilst methods define ways in which models can be synthesised and analysed to aid all stages of development. Although FOOD puts forward analysis and requirements capture, design and implementation strategies, these do not define an industrial strength software development method, which is beyond the scope of this thesis.

1.2 Context: Structured Development, Formal and Object Oriented Methods

The context of the work in this thesis is represented in the diagram in figure 1.2. The three main areas of interest, within software engineering, are structured software development methods, formal methods and object oriented methods. Within these areas we are most interested in formal object

oriented development (FOOD) and the role of the specification language LOTOS. These five areas are overviewed in sections 1.2.1 to 1.2.5, below¹.



1.2.1 Structured Software Development Methods

Structured software development techniques have emerged in response to the growing complexity of the systems being modelled. Constraints on the complexity of systems which can be handled have gradually moved from hardware to software. Consequently, programming languages have had to evolve to cope with much larger problems. In the early 1970s, the need for both methodological and formal approaches to programming were identified in a number of different texts, the best known of which are [40, 44, 127, 45, 105, 116]. High-level programming languages encouraged *structured programming*: the development of programs composed of (sub)programs composed of etc. . . ., combined with the ability to share complex data structures between programs (Modula-2 [128] is a good example of a language which encourages structured programming [18]). Structured programming leads to a natural functional decomposition of a problem. This is reflected in the large number of software design methods which place emphasis on functional structure. Good examples of these methods are found in [27, 90, 94, 36, 41, 51, 80]. A good overview of the differences and similarities between these methods is found in [17].

The functionally based structured software development methods placed emphasis on data flow modelling (in various forms), which shows the transformation of data as it flows through a system. The transformations are carried out by functions and design involves the repetitive division of functions into (sub)functions until the lowest-level components are simple enough to implement directly. This simple view of the functional divide-and-conquer approach does not do justice to the complexity inherent in many of the structured methods, but it is the underlying strategy in each of these techniques.

There are many reasons why functional approaches are in such wide use:

- Programmers think in terms of functions and so find the transition to structured analysis and design methods easy to make.

¹These sections give an overview of, and references to, some of the most important work related to the thesis. More comprehensive references are given in the appropriate parts of the main body of the thesis. It was considered counterproductive to give references to all work at this introductory stage.

- Functionally based methods were the first widely available and well documented techniques.
- The techniques offer good project management support, which plays a vital role in software development [49, 66].
- The methods are founded on graphical models, which appeal to the users.

However, as the rest of this thesis argues, approaches based on functional decomposition (data flow) are, in general, *inferior* to methods based on data structure. Further, the inherent informality in the most popular structured methods is not easily overcome.

1.2.2 Formal Methods

Many software engineers do not acknowledge the value of formality. This thesis is founded on the belief that formal methods are *just about* ready for transfer to general software development in industry. This belief is supported by a major study of the current state-of-the-art in formal methods [32] which concludes by stating:

“...formal methods, while still immature in certain important respects, are beginning to be used seriously and successfully by industry to design and develop computer systems ...”

This reinforces the statements made in [50, 71, 5] concerning the ever increasing importance of formality in the software development process.

There are a wide and varied range of definitions of *formal method* which can be found in the majority of texts concerned with mathematical rigour in computer science. (A wide range of *formal methods* are considered in [46, 115, 3, 96, 72, 19].) For the purposes of this thesis we propose the following definition:

A formal method is any technique concerned with the construction and/or analysis of mathematical models which aid the development of computer systems.

Formal methods are fundamentally concerned with *correctness*: the property that an abstract model fulfils a set of well defined requirements. This notion is addressed in a number of different texts and with respect to a number of different models, see [9, 10, 8, 15, 43, 34] for example.

A major problem when using formal methods in software engineering is that much of the recent research places emphasis on analysis rather than synthesis. The means of constructing complex formal models is often overlooked in favour of techniques for analysing models. In this thesis, emphasis is placed on the construction of formal models. Formal method engineers need to learn techniques for building very large, complex systems. Such techniques have been followed, with various degrees of success, by programmers. In particular, object oriented programmers have evolved techniques which have been successfully transferred to the analysis and design phases of software engineering. Where better then to look for aid in the construction of large formal models?

1.2.3 The Object Oriented Paradigm

The object oriented paradigm arose out of the realisation that functional decomposition is not the only means of structuring code: an alternative is to construct a system based on the structure of the data². Emphasis on data structure led to the encapsulation of functional behaviour within data entities: *objects*³.

Object oriented concepts were conceived in Simula [91], went through infancy in Smalltalk [58, 57] and could be said to be leaving adolescence, and approaching maturity, in the form of many different languages (for example: Objective C [31], C++ [106], LOOPS [7], Flavours [21, 88], CLOS [42, 73], Eiffel [84] and Common Objects [103]) and methods (for example: those of Rumbaugh [101], Coad and Yourdon [25, 26], Cox [31], Meyer [84] and Booch [13, 12]). [89] provides a good review of object oriented languages and methods with respect to object oriented analysis and design. Unfortunately, none of the well accepted methods provides a formal framework upon which the work in this thesis could be based.

1.2.4 Formal Methods and Object Orientation

There has been much interest in combining formal and object oriented methods. The research falls into two main categories:

- **i) Using Object Oriented Techniques To Construct Formal Models**

The success of object oriented techniques in software development has led to much interest in using the same techniques for building formal models. Much of this work centres on the definition of object oriented constructs, or the interpretation of object oriented concepts, in an existing formal language. Good examples of the type of work which has been done can be found in [24, 6, 33, 75, 100, 118, 56, 97, 81, 77]. This work has led to recognition of the inconsistent use of object oriented terminology, highlighting the need for a concensus of opinion. Further, much of the work shows the difficulties inherent in modelling object oriented behaviour in a semantic framework which was not designed for such a purpose.

- **ii) The Development of Object Oriented Semantics**

The lack of agreement on the meaning of object oriented constructs, reinforced by the informal semantics of most object oriented programming languages, has led many people to produce formal object oriented semantics, for example see [14, 130, 95, 123, 47]. The thesis by Wolczko [129] provides a more complete view of the technical issues, whilst Wegner [124] and America [1] examine the philosophical aspects. Much of this research has had a positive influence on the work in this thesis. However, the semantics examined were not deemed suitable for use in this work because they did not fulfil our three main requirements (apart from formality):

²Of course, there are programming languages which do not place emphasis on functional or data structure, but we do not consider them in any detail as part of this work.

³Two well known data-based software development methods which are generally accepted as not being object oriented are the quite similar approaches put forward by Jackson [69] and Orr [93]. These approaches are closely related to the object oriented paradigm in the initial analysis stages, but digress from the standard object oriented view as they approach implementation.

- 1) We require a semantics which agrees with our intuitive understanding of object oriented concepts, as recorded in chapter 2.
- 2) We require a semantics which, with a suitable syntax, is accessible to the customers for use during analysis⁴.
- 3) We require a semantics which can be used during all stages of object oriented development, in other words we need a wide-spectrum language.

Given these very specific requirements, it was necessary to define a new semantic framework.

The work in this thesis has a foot in both these areas, which share common ground. We develop an object oriented semantics (based upon a constructive, easy to understand, state transition system model). The abstract data type (ADT) part of LOTOS (*Language Of Temporal Ordering Specifications*), see [15, 112, 113, 68, 117], is used to implement the requirements models which are defined using this semantics. Then, *full LOTOS* (LOTOS with the ADT and process algebra parts) is used to model the requirements in a more concrete framework. In the thesis, LOTOS specifications are constructed using object oriented techniques and LOTOS is used to define a high-level object oriented semantics: these views are complementary.

In this thesis, emphasis is placed on semantic continuity. The object oriented semantics, based on the labelled state transition model, are present throughout development. LOTOS is used only to make the object oriented models more concrete, as development approaches implementation.

1.2.5 LOTOS

LOTOS is chosen as the object oriented requirements capture and design language because:

- Its natural division into ADT part (based on ACT ONE⁵ [48]) and process algebra part (similar to CSP [65] and CCS [87]) suits our need for semantic continuity from analysis to design: the ACT ONE requirements model can be incorporated within the full LOTOS design model.
- LOTOS has already been the subject of research with regard to its suitability for modelling object oriented systems and incorporating object oriented principles: for example, see [6, 118, 100, 81, 75, 33, 24].
- LOTOS is a wide spectrum language, which is suitable for specifying systems at various levels of abstraction. Consequently, it can be used at both ends of the design process.
- There is wide support, often in the form of tools, for the static analysis and dynamic execution of LOTOS specifications: for example, see [117, 61, 10, 92].

Although ACT ONE and LOTOS are prominent throughout FOOD, the main work in this thesis revolves around the principles rather than the languages used to implement the principles. Any abstract data typing language has the potential to implement the requirements models. Similarly,

⁴We argue, in later chapters, that this is possible only if the requirements models being used are constructive.

⁵In the remainder of this thesis the ADT part of LOTOS is referred to as ACT ONE, even though this identity is not quite precise.

any process algebra can be used to specify the internal and external communication models which are introduced in design. The problem of going from analysis to design (abstract types to more concrete processes) is made easier in LOTOS because of the way in which the two different parts of the language are integrated.

1.3 Formulation of an *Ideal Development Environment*

In an ideal software development environment the following requirements must be met:

- **Consistency**

There must be a consistency in the models of conceptualisation used throughout development. Further, there must be a coherent approach which can be applied consistently in different problem domains.

- **Correctness**

It must be possible to guarantee that the final *implementation oriented* design is *correct* with respect to the initial *customer oriented* requirements model.

- **Re-Use**

It is vital that re-use is prominent at all stages of software development, see [98, 52]. When constructing any engineered artifact it is desirable to use *materials* whose behaviour is well understood. In software engineering these *materials* are available in many different forms: low-level language constructs, medium-level predefined code components and high-level development methods.

- **Opportunism**

The software development method should not be too prescriptive. A developer should be able to do what seems best at any stage of development within well defined bounds, and the method should support this. The lack of opportunistic flexibility often constrains software development, see [119, 76, 62, 79]. Uncontrolled opportunism is not desirable, but there is no reason why software developers cannot be encouraged, by the method being employed, to both *craft* and *engineer* software.

- **Customer Awareness**

The analysis and requirements capture phases of software development should be *customer oriented*: it is generally agreed that customer communication is the most important aspect of analysis [67, 99, 114]. The successful synthesis of a requirements model is dependent on being able to construct a system as the customer views the problem [54]: requirements validation is not possible if the models cannot be communicated to the customer.

- **Implementation Awareness**

It is important that the design process can be targetted towards a wide range of implementation environments. In particular, the implementers must be able to make a correspondence between constructs in the implementation domain with constructs in the final design.

In industry, there are other requirements of a software development environment which, although important, are not listed above. We believe that meeting the six requirements, above, provides a framework upon which all other requirements can be met.

1.4 Formal Object Oriented Development (FOOD): Prototyping An *Ideal*

1.4.1 Fulfilling The *Ideal* Requirements: An Overview

Combining formal and object oriented methods provides a framework upon which the *ideal* development requirements can be met:

- **Object Oriented \Rightarrow Consistency**

The object oriented paradigm provides a conceptual consistency throughout the development process. It is proposed that one abstract model of object oriented concepts is maintained throughout analysis, design and implementation. Certainly, the concepts are then realised in more concrete terms as the models approach implementation, but the underlying abstract semantics are constant. Experience suggests that object oriented techniques are applicable in most problem domains [110], and consequently their use can be consistent from project to project⁶.

- **FOOD: The Road To Correctness**

There are three important aspects to the development of correct software:

- i) Validating Customer Requirements

A formal requirements model means that the customer does not have to deal with ambiguity, inconsistency or incompleteness. However, a formal model does not guarantee correct customer comprehension: the model can be wrongly interpreted⁷. An object oriented model is less likely to be misunderstood since it records the requirements in an intuitive way.

- ii) Correctness Preserving Design

Design is required to take an abstract, *customer oriented* model and produce a more concrete, *implementation oriented* model whilst preserving *correctness*. This can only be done when the two models are formally specified. The present state-of-the-art in formal methods cannot cope with the task of verifying any given formal design against any given formal requirements model. Consequently, this thesis advocates a transformational approach to

⁶Whilst this thesis agrees with the general applicability of object oriented techniques, we believe that other approaches are superior in particular problem domains. It is beyond the scope of this thesis to identify these areas.

⁷A further problem occurs when the customer misunderstands their requirements, rather than misunderstanding the requirements model. Such complexities are beyond the scope of this work, but we do believe that the process of constructing a formal object oriented requirements model reduces the risk of this occurring because of the resulting improvement in mutual understanding between customer and analyst.

design: design is defined as a sequence of transformation steps which reflect design decisions for moving from the abstract to the concrete. Each transformation must preserve correctness, i.e. the designers must be prepared to show that the latest design satisfies the requirements imposed by the previous stage. Correctness preserving transformations (CPTs) are a means of automatically guaranteeing *correct* design steps. It is important to note that there must still be a gap between the final requirements model and initial design (from *what* to *how*). However, a formal object oriented approach means that this gap can be bridged in a rigorous fashion.

- iii) Testing the Implementation

Presently, most implementation languages are not formally defined. Consequently, one cannot, in general, prove the correctness of an implementation with respect to a formal design. The formal object oriented approach does, however, make the testing of implementations much more rigorous: the code can be verified against unambiguous requirements and the object oriented conceptual consistency means that there is a structural correspondence between design and implementation. This promotes a compositional approach to testing. Further, the emphasis on re-use means that much of the implemented system has already been tested elsewhere.

- **Object Oriented \Rightarrow Re-Use**

The object oriented paradigm gives rise to two different types of re-use: composition and classification. Classes and class hierarchies provide re-usable components at all stages of development and at different levels of abstraction, see [83, 70]. Care, however, must be taken when inheritance is used as a code re-use facility [108].

- **Object Oriented Methods Support Opportunism**

There is an inherent flexibility in object oriented development:

- Object oriented development is both bottom-up and top-down, supporting composition and decomposition at all levels of abstraction. Developers, therefore, are not restricted to a simple repetitive analysis-synthesis-test sequence.
- The scalability of object oriented methods and consistency between all stages of development means that different parts of a system may be at different levels of abstraction without undue complication. Consequently, developers can easily move between system parts in a flexible fashion.

- **Customer Awareness**

The need for *customer oriented* models has already been emphasised in this chapter. Object oriented models certainly reflect the way in which customers comprehend their problem, but not all object oriented models are accessible to customers, particularly those with formal semantics. It is important that these models are presented in the form of *customer oriented* notations: in practice, this means providing graphical representations of the formal models [53]. The object oriented paradigm has a number of similar, but inconsistent, graphical representations

associated with particular development methods. These types of diagram can be provided in a formal framework without diminishing their ability to improve customer communication.

- **Implementation Awareness**

Formal methods can provide designers with enough power to target the designs towards a wide range of implementation languages and environments. Further, object oriented designs can be implemented in a wide range of programming languages, not just those which are object oriented.

This section has argued that a formal object oriented approach has the potential to provide an *ideal* software engineering environment. Figure 1.3 contains more details with respect to the components (models and methods) which are used, in this thesis, to construct a framework for the development of such an *ideal*. This prototype construction is certainly not the only (or *best*) way of providing such a framework, but it does show the feasibility of developing an industrial strength formal object oriented software development environment and illustrates the advantages in doing so.

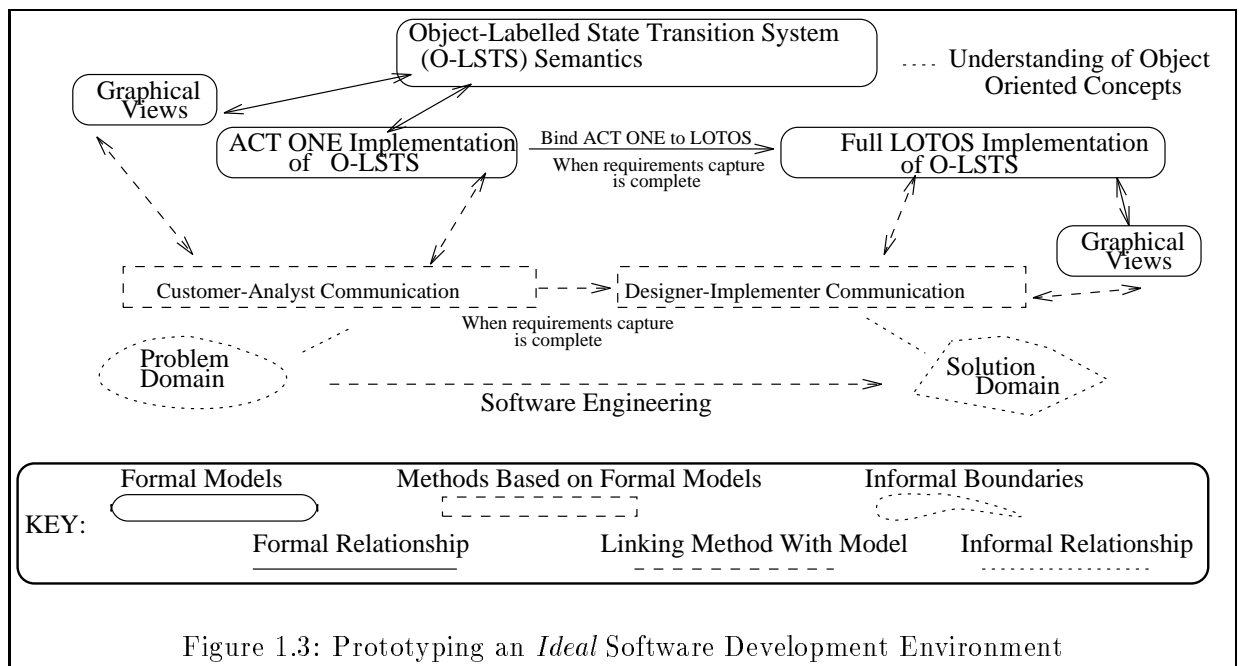


Figure 1.3: Prototyping an *Ideal* Software Development Environment

1.4.2 A Step-by-step Construction of the FOOD Environment

The main body of this thesis constructs a prototype of an *ideal* software development environment, based on the framework outlined in figure 1.3. The work naturally progresses from analysis to design and onto implementation. An overview of the structure and contents of each of the remaining chapters of this thesis is given below.

Chapter 2. Analysis: Modelling Problem Understanding

This chapter introduces formal object oriented analysis and the object oriented paradigm. An overview of software analysis methods places emphasis on problem domain understanding, customer

communication and requirements notation. A list of features which should be evident in a *good* analysis method are then identified. The object oriented paradigm is investigated as a means of providing a framework upon which all these features can be offered. The need for formal analysis models is introduced, and abstract data types (ADTs) put forward as a natural means of modelling classes of objects during analysis. This leads to an initial investigation of the relationship between classes and types. The chapter concludes with a preview of the formal object oriented analysis models which are developed in chapter 3.

Chapter 3. An Object Oriented Semantic Framework

This chapter formalises the understanding of object oriented concepts arising from the investigation in chapter 2. An abstract object oriented semantics is defined as a particular type of state transition system. From this simple basis the formal definitions of class, object, attributes, subclassing, composition, configuration and interaction are derived. The semantics are then syntactically sugared, placing emphasis on the object oriented concepts, to provide a formal object oriented analysis language which is accessible to customers and analysts alike. Graphical views of the object oriented properties, implicit in models defined using this language, are developed. An executable model of the analysis language semantics is provided by a translation to ACT ONE.

Chapter 4. Formal Object Oriented Analysis: The Practical Issues

This chapter examines practical issues which arise when analysing and synthesising the formal object oriented requirements models. Emphasis is placed on the construction of classification and composition hierarchies, with particular regard given to the difference between subclassing and delegation. Communication and interaction are considered at an abstract level: *what* rather than *how*. Other analysis issues, namely nondeterminism, exceptions and implementation freedom are also examined. A method, in the form of a high-level algorithm, is given for the development and validation of object oriented requirements models. Finally, the step from requirements to design is previewed.

Chapter 5. Formal Object Oriented Design (Using LOTOS)

This chapter begins by introducing design and highlighting the importance of learning from different design areas. Software design is then considered, with a focus on object oriented methods. This is followed by an overview of LOTOS and its suitability for modelling object oriented systems. Then, the means of going from an ACT ONE requirements model to a full LOTOS design is examined. The importance of maintaining correctness across the design process is stated. With this in mind, the notion of correctness preserving transformation (CPT) is introduced. Finally, a small set of CPTs are defined to correspond to decisions that are commonly made during object oriented design. Design is then defined as the process of transforming a *customer oriented* requirements model to an *implementation oriented* model, using CPTs wherever possible. The target implementation model is shown to have a great influence on the design decisions.

Chapter 6. Object Oriented Program Derivation

The implementation of high-level object oriented LOTOS designs is considered. Many different target implementation environments are examined, not just those which are object oriented, with respect to their suitability for use in the final implementation stage of FOOD. It is argued that object oriented programming languages provide the easiest targets at which object oriented designers can aim. Eiffel [84] is chosen, from a wide range of object oriented programming languages, to illustrate the implementation process. The importance of having a thorough understanding of programming language semantics is stated. The informal Eiffel semantics are reviewed and a methodological, but informal, technique for generating Eiffel code from object oriented LOTOS designs is proposed. Finally, the future generation of concurrent implementations is identified as an area which is well supported by FOOD.

Chapter 7. Formal Object Oriented Development: A Case Study

This chapter puts all the theory from chapters 2 to 6 into practice. A small banking network system is developed using the analysis, requirements capture, design and implementation models and methods proposed by this thesis. The case study does not utilise, or consequently test, all aspects of FOOD, but it does show that the method has the potential for use in real software development projects. To conclude the chapter, a list of the more important lessons arising from the case study is given. The most important lesson is that much more work remains to be done.

Chapter 8. Conclusions

This chapter concludes the thesis by reviewing the initial objectives, showing how the thesis meets these objectives and identifying areas of further work arising out of the thesis.

1.5 Contributions of the Thesis

The main contributions of the thesis are: the philosophy and reasoning behind a formal object oriented development strategy, the object oriented mathematical models resulting from this reasoning, the software engineering methods which utilise the models in a consistent and coherent fashion (collectively called FOOD) and the preliminary case study which shows the effectiveness of these methods.

1.5.1 FOOD: The Philosophy

The philosophy upon which FOOD was developed is a major contribution of the thesis:

Formal and object oriented methods are complementary. *Correctness* is the most important property of software. Formality is the only means of guaranteeing *correctness*. The complexity of constructing software models (at all levels of abstraction) can be managed using object oriented techniques. Formal object oriented techniques help to bridge the

informal gaps at either end of software development. Formal object oriented development is *software engineering* in a pure form.

1.5.2 FOOD: The Models

The need for precision when defining object oriented concepts leads to the development of a number of formal models. An abstract object oriented semantics is developed to form the basis of our understanding of all object oriented models (from analysis to implementation). *Customer oriented* analysis and requirements models are defined, together with a formal means of stepping from requirements models to initial designs. Correctness preserving transformations are defined to reflect object oriented design decisions. The informal step from final design to an implementation model is strengthened by a rigorous investigation of the semantics of Eiffel (the chosen implementation language). The way in which LOTOS is used throughout FOOD is both original and effective.

1.5.3 FOOD: The Method

The emphasis during software development using FOOD is on rigour, re-use and opportunism. A skeleton method is provided for the requirements capture, design and implementation stages. This method is not yet strong enough for industrial use⁸, but there is potential for either integrating FOOD with other more commercial techniques or extending FOOD with less technical but more practical constructs. The method has been used in a trial case study to illustrate its effectiveness.

Contribution Summary

As a whole, the thesis offers a clear and concise statement of the problems inherent in software development, together with proposals for solutions to these problems which are based on the integration of formal and object oriented methods. A framework (FOOD) is developed for the implementation of these solutions.

⁸We believe that a method should evolve from experience of using the models.