FAD: A FUNCTIONAL ANALYSIS AND DESIGN METHODOLOGY

A THESIS SUBMITTED TO
THE UNIVERSITY OF KENT AT CANTERBURY
IN THE SUBJECT OF COMPUTER SCIENCE
FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY.

By
Daniel John Russell

August 2000

This thesis was supervised and approved by:

Dr. S. J. Thompson

Examiners:

Dr. I. MacCallum (External)

Dr. S. Kent (Internal)

To Mum, Dad and Anita

Abstract

This thesis presents the functional analysis and design methodology FAD. By functional we mean that it naturally supports software development within the functional programming paradigm (FP).

Every popular methodology has a graphical modelling language which presents various pictorial representations of a system. FAD's modelling language provides the typical elements of functional programming, types and functions, plus elements to support modular development such as modules, subsystems and two forms of signature which specify an interface or a behavioural requirement. The language also includes relationships and associations between these elements, and provides simple representations of functional designs. The methodology has an integrated set of techniques which guide the development of an implementable solution from the deliverables of requirements engineering. FAD's data dictionary provides an organised repository for entities during and after development.

The thesis thus provides a development medium which has been hitherto absent from the functional programming paradigm.

Acknowledgements

I would first of all like to thank my supervisor Dr. Simon Thompson who, if you will pardon the pun, was a model supervisor. When asked by some of my colleagues in a negative tone, to "tell me about your supervision", I could only disappoint them and say it was excellent. I am indebted to Phil Molyneux who some years ago encouraged me to take a copy of Bird and Wadler's *Introduction to Functional Programming* on holiday to India. It's strange, I loved it but it was not that popular with my fellow travellers! In addition, Phil has been a constant source of information, and an enthusiastic advocate for the beauty of functional programming. Other colleagues who have encouraged me all the way include Barry Walters, Sue Preston, Eva Smith, Huw Morris, Charles Blankson, and Phil Samouel. My weekend friend Stavros Kalafatis has been positive when I wanted him to be, and assertive at other times. Can I come out of my room now Stavros? Many of my friends have provided endless support. Jane Hillston many, many years ago encouraged me to embark on such a venture, and has always been there to listen when necessary, and provide advice when appropriate. Richard Burke has been brilliant! With periodic encouraging advice like *just get on with it*, he has been with me all the way and has simply been a friend. Judy Arroyo has been another great friend, although I think her motivation for encouraging me to finish is so that I can go and visit her in the States. My last thanks must go to my family. My mum was always interested in what I was doing and provided love all the way. My dad, brother and sister have continued to provide love, support and encouragement. And finally, my wife Anita - if she remembers who I am - has carried me through this process. She has been simply wonderful.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Developing well-designed software is difficult; developing poorly designed software is a lot easier. Anybody with some programming skills can produce programs that satisfy some basic stated requirements. Problems may arise when the code is passed to somebody else to maintain or one attempts to reuse elements of the program or the program itself. Can segments of code be used independently of the program for which they were originally developed? What are the major data structures of the system and how are they constructed? What is the major functionality supported by the system? If one makes changes to a certain piece of code what effect will this have? If the answers to these types of questions tend to be negative or difficult to determine the software is probably poorly designed. Unfortunately good design does not flow naturally from the fingertips of programmers.

Good design requires support.

Some support is provided by programming languages. Object-oriented (OO) languages provide mechanisms for developing software built on units that encapsulate their state and provide an explicit interface for potential clients. Thus, *if practised sensibly*, one can develop software where changes have a local effect, significant elements are reusable and can be reused independently. However practising sensible OO development is not a trivial process. OO developers can seek help from a plethora of OO analysis and design methodologies and various rules, heuristics and laws which provide substantial guidance and support.

Programmers who use imperative languages have for some time been encouraged to adopt a structured programming approach supported by various structured analysis and/or design methodologies. In common with object-orientation, the development paradigm is consistent within all the media of development.

Therefore when deciding which software development approach to adopt, the support afforded by either of these paradigms may have a significant influence.

It is certainly the case that the functional programming (FP) paradigm has been relatively unsuccessful in competing in the marketplace with the object-oriented and structured paradigms. Although one can enumerate an ever growing list of 'real world' applications [141, 123] written in functional languages, in comparison to the other paradigms it is relatively insignificant. Proponents of the FP paradigm can present several good reasons why it should be adopted in preference to its competitors. For example, the higher-order and typed (HOT) characteristics of modern FP languages have certainly influenced the design of non-FP languages such as Java. However one can present a range of historical (programming and non-programming related) cases where the fittest didn't always survive, and therefore, there is clearly a need to focus on the possible reasons for this slow uptake, and resolve as many of the problems as possible.

Wadler addresses this issue in his paper *Why no one uses functional languages* [142] where he includes among the historic reasons: that functional languages are often under active development, the non-compatibility with existing code written in other languages, the relative lack of language libraries to support software reuse, and the dearth of software development tools including software development methodologies which support implementation in a functional language.

The Haskell community has recently defined Haskell 98 [100], a stable version of Haskell allowing potential users to adopt it without fear of imminent change. Haskell is now available in various implementations including the interpreter Hugs [67], GHC [104] and the University of Chalmers's HBC compiler [55]. Standard ML [88] is evidence of similar developments within the ML community.

Compatibility with code written in other languages is addressed through recent work on H/Direct which allows a functional language, Haskell, to inter-operate with C and COM, and allows a Haskell component to be wrapped in a C or COM interface [42].

Software libraries are being developed to support a variety of application domains in the functional paradigm. For example, TclHaskell is a library of functions for writing platform independent, graphical user interfaces in Haskell [135] and FranTk, a declarative library for building GUIs in Haskell [48].

There has also been a lot of excellent work on developing correct programs [124] and in the complementary areas such as compiler efficiency [128, 5]. What has been lacking however, is a parallel focus on the development of certain support materials.

Some profilers have been developed [121], a lot of research is focusing on improving error messages [40, 10] and a small amount of work has been done on debugger development [136], but software development methodologies to support functional programming-in-the-large are virtually non-existent.

Particular languages such as Erlang [6] are accompanied by development environments, but for functional programming to be taken seriously, and not to be viewed as a toy to be either played with in academic departments or research groups, or whose only use is as an executable prototyping tool, then we need to support development using any functional language with language-independent but paradigm-dependent analysis and design methodologies and their accompanying CASE tools. Functional programming's competitors have not only been doing this for some time but they have also been doing it with evident success.

1.2 Graphical Notation

A functional analysis and design methodology requires a modelling language whose elements deliver natural models of functional programming designs. A graphical language is preferable since one is focusing on modelling abstractions rather than algorithmic details. Graphical representations of functional programs have been used for sometime albeit informally. For example, in Figure 1 we present a box-and-arrow (or purely functional data flow) diagram of a function which returns the sum of the integers within a stated range [111]. Jeffrey [45] has written a Java applet *Flow Graph Editor* in which one can create such diagrams.

One would be hard pushed to claim that the diagram is easier to understand than the equivalent code written in Haskell which also includes explicit type information.

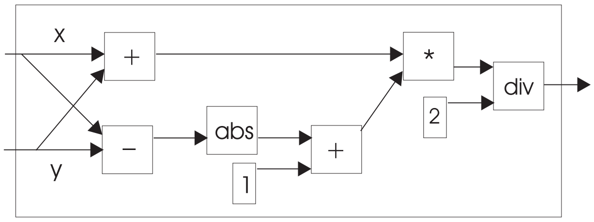

Figure 1: Box-and-Arrow Diagram

```
sumBetween :: Int -> Int -> Int
sumBetween x y
        = let sumG  = x + y
              diffG = x - y
              size1 = abs diffG + 1
          in  sumG * size1 `div` 2
```

Cardelli [24] and Reekie [112] describe notations for *visual functional programming languages* in which functions are defined graphically. However, once again the focus is on representing algorithms rather than abstract models of programs.

1.3 Overview of the Thesis

This thesis presents an analysis and design methodology which supports software development in the functional programming paradigm. The methodology uses a modelling language which supports the elements of functional programming and naturally models functional designs.

Chapter 2, *Object-Orientation*, provides a description of the OO paradigm, with an emphasis on the features which significantly affect software development. We chose to focus on OO rather than the structured approach since OO is certainly the predominant paradigm for developing new software. The OO features are highlighted both within the languages of the paradigm and its methodologies. We argue that adopting a *packaged approach* using a methodology and implementation language of the same paradigm should improve the development process and remove a lot of *accidental complexity* due to having to switch from one paradigm to another.

Chapter 3, *Functional Programming*, provides a similar description of the functional programming paradigm, drawing comparisons where appropriate with OO. FP is a significantly different approach to developing software, and therefore, requires significantly different methodologies to support the process.

Chapter 4, *Analysis and Design Methodologies*, gives a brief description of methodologies, their modelling languages and the techniques which together deliver a methodology.

Chapter 5, *FAD Modelling Language*, describes the modelling language of the *Functional Analysis and Design Methodology* (FAD). We describe each of the elements of the language which are used to model FAD designs. In the first section a case study is described which provides a major example upon which the language and techniques of the methodology can be illustrated. The case study is the development of an automated *football results processing system.* A data entry clerk enters recent results and can request the generation of various football-related information. The case study was chosen because it is large enough to illustrate the application of the methodology but small enough to comprehend fully. Each element of the language is accompanied by its graphical notation. The syntax and semantics of the methodology's diagrams are presented in an informal manner.

Chapter 6, *FAD Functional Designs*, presents illustrative examples of the ease with which functional designs can be modelled in FAD.

Chapter 7, *FAD Methodology*, describes the methodology as a list of tasks. The presentational style is linear, within the phases *analysis* and *design* but we emphasise that the methodology should be practised as an iterative and incremental process. The methodology includes several techniques, many of which are used within more than one task. Each technique is describe in terms of its required inputs, deliverables and activities. The deliverables are typically presented in diagrams and recorded in various documentation.

Chapter 8, *Data Dictionary*, presents an overview of the data dictionary which delivers an efficiently organised medium for storing entities. This supports the development of designs built on existing entities, and the discovery of common abstractions. Each entity is recorded in a *description document* which provides keys to their storage location.

Finally, Chapter 9, *Summary*, summarises the thesis and lists its key contributions.

Future research and development requirements are presented including the need for CASE tools to support the use of the methodology.

Throughout this thesis the names of case study entities - types, functions, signatures, modules, subsystems, files and projects - are presented in `teletype`, as is implementation language code. All functional programs in this thesis are written in Haskell 98. Object-oriented models are developed in UML [16]. Each FAD technique is introduced in *italics* which are also occasionally used for *emphasis*. Non-code example names are written in **bold** font.

Chapter 2

Object-Orientation

Much has been written about object-oriented (OO) software development. It has been variously described as evolutionary, revolutionary or both when compared to its predecessors. Whichever is the case it has been successful when measured in terms of the number of job adverts requiring skills in particular OO languages or OO development in general. The sizeable number of object-oriented languages (OOLs) and object-oriented analysis and design methodologies (OOADMs) are supported by innumerable texts, language implementations and CASE support tools. There is a wide variety of texts on specific languages such as Java [36, 148], Smalltalk [56, 51], Eiffel [115], C++ [133, 85, 86], and JavaScript [44], and equally prolific are the texts on particular OOADMs including the Booch Method [15, 82], OMT [120], OOSE/Objectory [64], Fusion [30] and more recently development approaches supported by the modelling language UML [16, 46, 109]. CASE tools include Rational's Rose [33, 34] which supports Booch, OMT, and UML notation, and OOA$Tool^{TM}$ and OOD$Tool^{TM}$ which support Coad/Yourdon's OOA/OOD methodologies [28, 29].

The ubiquity of the object-oriented paradigm in its various guises leads one to conclude that the argument often-made that the object-oriented (OO) approach is the most natural and robust way to develop software, through its focus on managing dependencies, is certainly not vacuous [82, 29]. Budd [21] provides a quote from Newsweek which gives an insight into the reasons for the popularity of object-orientation

> Unlike the usual programming method - writing software one line at a time

- NeXT's "object-oriented" system offers larger building blocks that developers can quickly assemble the way a kid builds faces on Mr. Potato Head.

There are however other reasons for OO's popularity. Software can be developed from its inception, through to implementation and beyond, within the OO paradigm. That is, one can adopt a single packaged approach to software development aided by a significant number of modelling languages, methodologies and CASE tools. Object-orientation is presented as a software development philosophy and not simply a term for classifying a collection of implementation languages. Each member of the paradigm supports, at a certain level of abstraction, a consistent approach to software development.

In this chapter we present an overview of the paradigm with an emphasis on those features that have a major impact on software development. In the first section we describe the features of the paradigm that have a significant effect on software development, and in some cases, distinguish it from other paradigms. These include: objects and classes as the fundamental building blocks of the paradigm; inheritance, composition and aggregation as the essential glue for construction of programs; and, inclusion polymorphism, dynamic binding and subtyping, which provide significant support for reuse. Where appropriate we will provide the motivation for the introduction of a feature and draw comparisons with its predecessors such as structured development. Section 2.2 presents an overview of current OO languages highlighting their similarities and differences. We discuss single and multiple inheritance, and the various approaches to encapsulation. This is followed in Section 2.3 with a review of existing analysis and design methodologies and modelling languages. We conclude with some brief remarks on the benefits of analyzing, developing, and implementing software wholly within the OO paradigm. Where possible we will endeavour to introduce notation before using it, but will undoubtedly on occasion be unable to uphold this principle.

2.1 The OO Paradigm – Motivation and Features

The object-oriented paradigm is evident in a collection of programming languages, software development methodologies and database systems. There are actually two OO paradigms. The 'classical' OO paradigm which refers to the class/object approach, and the 'delegation/prototyping' OO paradigm where objects delegate responsibility to other objects, as in the languages Self [137] and JavaScript [44]. We will primarily focus on the classical approach since most OO languages and OOADMs adopt this paradigm.

Object-Orientation: Evolution or Revolution

Object-orientation is described by some of its proponents as both an evolution and a revolution [21]. It is an evolution because it follows on naturally from earlier software development approaches. OO has addressed the various problems with the structured development approach. These include its lack of support for modularity, the potential for data insecurity due to the separation of data and functionality, and the higher priority given to the solution domain rather than the problem domain. However the foundations of most OO languages remain imperative in nature. One must not forget of course that structured programming was itself a reaction to problems with its predecessors [38, 35].

OO is regarded as revolutionary since it adopts an approach to modelling a software solution that is significantly different from its predecessors [82]. Where the structured approach focuses on data and processes that are universally accessible, OO describes them through abstractions which hide their details, and instead presents an explicit interface for any potential clients. Although structured programming is sometimes referred to as a predecessor of OO they were actually mooted at the same time [35]. However structured programming was easier to put into practice due to the availability of appropriate languages.

Booch [15] and others disagree with this revolutionary emphasis, and argue that OO simply reflected developments in various fields of computer science in the early 1970s. Objects were introduced to deal with the increasing complexity of software systems. For example, database technology introduced the idea of the entity-relationship approach to data modelling [119, 26] where a system is described as a set of entities, their attributes and relationships. Entities in entity-relationship diagrams (ERDs) are similar to classes without the operations.

Object-Orientation: Approach to Software Development

OO software is developed through a collection of interacting, extensible, abstractions which host their own state, provide mechanisms for manipulating the state, and deliver an explicit behavioural contract to other abstractions. That is, OO delivers an architecture within which control is decentralised to a focused collection of entities. The OO software engineering philosophy is to be problem-centred rather than solution-centred. One should therefore describe and model the problem in terms that are familiar to the system user and not to the computer professional. That is, one models tangible and intangible problem elements as abstractions in which data and process are combined.

Systems are developed through extending these abstractions and declaring other associations to support communication between the abstractions. The communications are controlled via an explicit interface. That is, each abstraction knows enough and no more about any abstraction with which it communicates. This is achieved through building the abstractions guided by the complementary concepts of *abstraction* and *encapsulation*. Pooley and Stevens [109] summarize these terms in the following manner.

> Abstraction is when a client of a module doesn't need to know more than is in the interface. Encapsulation is when a client of a module isn't able to know more than is in the interface.

Thus, OO is explicitly modular, encourages information hiding through encapsulation of state and functionality, and if practised effectively should minimise maintenance costs and maximise reuse. These are not characteristics of object-orientation's historic competitors. Action-oriented structured development is procedure-driven and thus supports techniques for procedure development. These include algorithmic decomposition through the stepwise refinement of procedures, and building algorithms through the three constructs: *sequence*, *selection* and *iteration* [12]. Although adopting a structured approach should result in effective procedural code, it provides limited support for the development of complex systems and certainly no support for developing models which can be naturally implemented in an OO language. Structured programming is supported by methodologies including SSADM [41] and SA/SD [152, 153]. SSADM, in common with most structured programming methodologies, emphasizes three views of a system: structural, functional and dynamic, each supported by graphical representations in the form of *logical data structure diagrams* (or entity-relationship diagrams), *data flow diagrams* and *entity life history diagrams*. Although the structured approach recognises the importance of describing the data in the system through *entities* and their attributes as first described by DeMarco [37], and also supports entity subtype/supertype relationships, each entity has no behavioural characteristics and is acted upon by external procedures and functions.

The models produced through the adoption of structured methodologies are most naturally implemented in various imperative languages such as C [72], and COBOL [130]. Although data-driven design methodologies such as JSD [134] (Jackson System Development) do promote more of a problem focus, where the structure of the solution mirrors the structure of the data being processed, they still encourage a structured approach to algorithm development and lack support for modularity and information hiding.

Object FAQ [92], a web site which provides answers to frequently asked questions regarding object technology and object-orientation, presents the following motivation for the introduction of object-orientation.

> Modelling in analysis and software design and languages for programming originally focused on process. But many metrics and results indicated the process approach was problematic and a limiting factor in what could be achieved, perhaps by several orders of magnitude, which led to the software crisis [14]...The inclusion of objects to better represent concepts and process offers a superior capability that can be viewed as an improvement over the older (structured) techniques, or as a totally reengineered breakthrough advance resulting from philosophical inquiry and methodological improvement, the latter in terms of both pedagogy and pragmatics.

In common with any paradigm there is some debate regarding what constitutes object-orientation. Cardelli and Wegner [25] use the following equation in describing OO languages.

$$\text{object-oriented} = \text{data abstractions} + \text{object types} + \text{type inheritance}$$

This equation describes OO languages as extensions of procedure-oriented (or imperative) languages which support data abstractions, collecting objects with a common interface (type), and construction of a new interface through inheritance. If one removed the last operand, the right hand side of the equation would describe *object-based* languages. Coad [28] provides a different but similar equation whose right hand side is:

classes and objects + inheritance + communication with messages

This equation describes the significant majority of OO languages which create objects through the instantiation of a class. These languages are typically referred to as *class-based languages*. This equation also indicates that objects are a mechanism for encapsulation, where behaviour is implemented through objects communicating via messages.

In the following sections we describe the essential features of OO and how they influence software development within the paradigm. They include objects and classes, inheritance, composition and aggregation, and inclusion polymorphism and subtyping. The first and most obvious feature is the use of objects as software building blocks.

2.1.1 The Building Blocks – Objects and their Classes

An object is a mechanism for encapsulation and abstraction. It hosts state, the methods which act on the state, and an interface to the object for any potential clients. Thus an object normally has a number of named *attributes* or variables representing its state, and a collection of *methods* that implement the behaviour required of the object. A subset of these methods and attributes, typically empty in the latter case, will be specified in the object's interface. Each object actually presents two interfaces sometimes referred to as the *public* and *protected* interfaces. The public interface is the interface presented to all potential clients and we will refer to this simply as the interface. The protected interface is presented to clients from within the object's inheritance hierarchy. We describe inheritance and clarify this distinction in Section 2.1.2. Each object has a unique identity which is independent of the values of its variables.

OO development emphasizes the separation of *what* from *how* through encapsulation and abstraction. A client module wants to know what it can do with a server module, and not how the server supports this functionality. An object's interface specifies the *what*, with the *how* largely inaccessible to clients. One can therefore quite naturally adopt Parnas's *information hiding* modular development criterion when developing OO software [95], using objects as the mechanism for information hiding.

Class-based versus delegation-based

Most object-oriented languages are class-based and thus sit within the classical OO paradigm. Objects are created through the instantiation of an abstraction called a *class* which defines each of its object's attributes, methods and interface. It is not uncommon to equate an object's class and its *type*. However the object-oriented view of a type is as a *behaviour specification*. Since an object's interface specifies behaviour, every object of a class shares a type. However, objects of other classes may also support the same behaviour and therefore have the same type. In addition, an object may support a subset of the behaviour declared in its class, and thus an object can have more than one type. The relationship between classes and types are generally linked to the inheritance mechanism that we describe in Section 2.1.2. Thus the class `X` defines objects with a single constructor method (also called `X`), a single attribute `i` of type `int`, and two other methods `method1` and `method2`. The three methods together form the interface specified by the class as indicated by the keyword `public`. The keyword `private` indicates that the attribute is not part of the interface. The object `xObject` is an instantiation of the class `X`.

```
class X {
        public X(int n){i=n;}
        public int method1(Y y){
                return (i*y.get());}
        public int method2(){
                return i;}
        private int i;
}
X xObject;
```

In dynamically-typed, class-based languages such as Smalltalk, an object's class is simply used for object implementation and not to provide type information. In statically-typed languages like Java, a class both provides object implementation details, including mechanisms for object construction, and type information through the declared interface.

One can decouple interface declaration from implementation declaration through only providing specifications and no implementations in a class declaration. Implementations can be added to a class which inherits from an 'interface-only' class. A full description of the inheritance mechanism is presented in Section 2.1.2. A class which provides either no implementations or an incomplete set of implementations is referred to as an *abstract class* or alternatively an *abstract base class* or *abstract parent class*. The latter two names signal their use in class development through inheritance. Since an abstract class provides an incomplete blueprint for an object, there are no objects of the class. However one can use abstract classes to declare an interface that will be supported by any object whose class inherits from the abstract class. Thus the class `X` could inherit from the abstract class, `AbstractX`. The keyword `abstract` indicates that the class is abstract and therefore has no instance objects. An `abstract` method does not have a method body, and therefore requires definition in any subclass.

```
abstract class AbstractX {
        public X(int n){i=n;}
        public abstract int method1(Y);
        public int method2(SubY y){
                return i;}
        private int i;
}
```

JavaScript and Self are OO languages which are not class-based. These are *delegation/prototyping* languages where object prototypes are used as the mechanism for the creation of new objects with extended behaviour. These are created through the addition of methods and/or attributes to those provided by the prototype object. This form of OO is sometimes referred to as *single hierarchy* since one simply has a hierarchy of objects (and no hierarchy of classes). Languages of this paradigm support both static and dynamic inheritance which we will discuss in Section 2.1.2.

Message Passing

Communication between objects is marshalled via their public interfaces. Budd [21] presents his first principle of object-oriented problem solving as

> action is initiated in object-oriented programming by the transmission of a *message* to an agent (an *object*) responsible for the action.

That is, a message is passed to an object, where the message includes information about which method to call and with which arguments. The object is responsible for invoking the method that satisfies the request. The behaviour of an object may depend both on the method's parameter values and on the values of the object's attributes. That is, it is not unusual for behaviour to be dependent on the state as it is in imperative systems. The difference is that the state is typically local rather than global. In functional programming systems, behaviour depends solely on a function's inputted values.

Ideally one should be able to send a message to any object capable of invoking the appropriate method. In practice, most OO languages are statically-typed which imposes constraints on which objects can receive a message. Whatever the typing mechanism method invocation is controlled by the object receiving the message. The route of message passing between objects has a significant effect on the amount of coupling between objects. The Law of Demeter [77], named after an object-oriented programming tool, provides guidance on the development of interacting objects. It states that an object, in response to a message, should only send messages to:

- the object itself or one of its attribute objects;
- objects created due to the message; or,
- an object provided as an argument to the message.

The tool will check whether a program conforms to the law.

The following section presents an overview of the OO mechanisms for developing software using objects and classes as the basic building blocks. These include attribute objects and objects as arguments alluded to in the Law of Demeter.

2.1.2 The Glue

In this section we describe various mechanisms for building OO software. These include inheritance, attribute objects and objects as arguments to methods. It is clear that objects and their classes provide a mechanism for modular software development guided by the requirements of encapsulation and abstraction. What distinguishes object-orientation from abstraction (or object) based development, which is supported by languages such as Modula-2 [150], is inheritance [15]. This is the primary development mechanism used within the object-oriented paradigm. It is a mechanism that, for better or worse, supports a range of use semantics including interface reuse, interface extension, and implementation or code reuse. Before describing other development mechanisms, we describe the various forms of inheritance.

Inheritance

The verb *to inherit* has two transitive definitions [31]

> to receive by legal descent, as heir or,
>
> to derive from parents

and a single intransitive definition

> to succeed as heir.

It is the second of the transitive definitions that best describes inheritance within the classical OO paradigm. A parent class is a class from which another class derives some of its features. Each class-based OO language either supports *single inheritance*, where a class can only inherit from a single class, or *multiple inheritance*, which supports multiple parent classes. Inheritance within the delegation/prototyping paradigm, links an object to a list of objects to which it delegates some of its responsibilities.

The terms 'parent class' and 'child class' are accepted terminology within the classical OO paradigm [16]. They are also referred to as a *superclass* and *subclass*. In fact, both the verb and the inheritance relation are transitive. That is, if the class **A** inherits from the class **B**, and **B** inherits from the class **C**, then **A** inherits from **C**. To take the parental metaphor one step further, **C** is a grandparent of **A**. Thus when using a class-based language one develops a *hierarchy* of classes linked through inheritance.

A class `Child` which inherits from a class `Parent` can adopt the attribute and method specifications, any attribute and method implementations, and the interface of the class `Parent`. If the class `Parent` is abstract then any non-implemented methods can be implemented in the `Child` class. Any implemented method of the class `Parent` can either be adopted or overridden by the class `Child`. An object of the `Child` class typically has special privileges in regard to access to entities of an object of the `Parent` class. These access rights are declared in the *protected interface* of the `Parent` class which is typically the public interface of the class plus some attributes which are hidden from general clients. We illustrate inheritance using the classes `X` and `AbstractX` referred to earlier in this chapter. `X` inherits the attributes and methods of `AbstractX` and provides an implementation for `method1`.

```
abstract class AbstractX {
        public X(int n){int i=n;}
        public abstract int method1(Y);
        public int method2(SubY y){
                return i;}
        private int i;
}
class X extends AbstractX {
        public int method1(Y y){
                return (i*y.get());}
}
```

The mechanics of interface declaration are language-specific, some of which are presented in Section 2.2.

Statically-typed, class-based OO languages only support static inheritance, or inheritance declared at compile time. Smalltalk, a dynamically-typed, class-based language and delegation/prototype languages support both static and dynamic inheritance. That is, one can create new forms of objects through inheritance at run time.

Every object of a class presents to clients the interface declared in the class. They can also present the interface of any ancestor class. Hence, two objects can have different classes but support the same behaviour as described by an interface. They thus have the same type. Thus an object can have more than one type, and a type be exhibited by objects of more than one class. In statically-typed, class-based languages, each variable is declared with an explicit class which states the variable's type. The variable can then be assigned any object of the stated class or its subclasses. In dynamically-typed languages, a check to determine if an object's class supports a required interface is performed at run time, and therefore one is not constrained to use objects of classes within a particular inheritance chain.

Inheritance is object-orientation's primary mechanism for reusing existing entities. Since an object has three rôles, a host of a set of attributes which make up the object's state, state manipulation through its methods, and access control through an interface, inheritance can enable reuse of any combination of these. Thus a child class could inherit only an interface from a parent class if that is all the parent class provides. Alternatively a child class could inherit attributes, functionality and an interface from its parent. This overloading of the inheritance mechanism can be viewed as both a positive and negative feature. It is positive simply because it is overloaded, and thus one can achieve multiple versions of reuse with the same mechanism. It is however a negative feature, since the semantics of a particular application of inheritance is a function of the characteristics of the parent class and child class, and not of the inheritance mechanism itself. Budd [21] presents a comprehensive list of the various forms of inheritance.

The combination of multiple rôle abstractions, and development through extension has important implications for software development within the paradigm. One is required in some sense to 'see the future' when modelling a collection of classes. Several questions need to be answered which include:

- Will the class's interface ever be reused without its implementations?
- Will I need a class with a subset of the functionality of the class?
- Will I need a class with more functionality than the class but less than another class that is being developed through inheritance?

Many texts on object-orientation devote substantial space to warnings about overuse or misuse of inheritance, often describing alternative designs available to the developer [49]. Although one can reuse code through inheritance it is generally accepted as bad practice since it breaks class-based encapsulation. A child class that reuses implementations from a parent class is tightly coupled to the parent class and, therefore, any change to implementations in the parent class could potentially have an effect in the child class. In addition, program correctness can be difficult to determine since an object's response to a message may be a method declared in an ancestor class.

Meyer's *design by contract* [84] has addressed this issue through the introduction of some formal rules of practice. These rules give formal guidance on the behaviour of a method, and on the development of overridden methods in subclasses. The rules require that a method should be accompanied by one or more preconditions (*require clauses*) on input values and host object attribute values, and state-related postconditions (*ensure clauses*). Methods which are overridden in child classes must have require clauses that are no more constraining then their ancestors, and ensure clauses which are no less constraining. Design by contract makes explicit the need for behavioural consistency between classes and their subclasses, where overridden methods in a child class preserve the behavioural characteristics of their parental counterparts. Design by contract is supported by the OO language Eiffel, and by the modelling language of BON (Business Object Notation) [143], but it is not generally supported by OO languages or OOADMs.

In summary, although inheritance provides a useful and natural medium for reusing interfaces and implementations, it can result in software built on tightly coupled modules, which is poor modular design. In addition, the reliance on inheritance for class and object building requires the developer to foresee any potential future developments, which makes iterative development difficult. Gamma [49] points to the problem of inheritance hierarchies continually having to be rearranged as the prime motivator of his work on reusable design patterns. In the following section we describe alternative mechanisms for developing OO software.

Other OO Glue

A developer using the object-oriented paradigm can draw upon other non-inheritance object/class associations during system development. They include passing objects as parameters to methods, and objects as attributes of other objects.

The functional programming paradigm and the object-oriented paradigm differ in which constructs are *first-class* where first-class constructs are those that can be treated like any data value. Functions are first-class in functional programming and therefore can appear in data structures and be supplied as arguments to functions. Objects are first-class in the object-oriented paradigm and for example, can be passed as parameters to methods of other objects. In a pure OO language with no non-object values, only objects (or in certain cases classes (see Section 2.2)) can be passed as arguments to methods.

An alternative to adopting another class's behaviour through inheritance is to build objects which 'include' other objects as attributes. There are two general forms of object attribution. The first is where the object attribute is declared in the host object, and thus is dependent for its existence on its host. This is sometimes referred to as *composition* or *composite aggregation*. In the second form the attribute object could be declared independently of any potential host object, which associates itself with the attribute through a variable which references the used object. Thus the attribute object may be used in this manner by several other objects. This form of object attribution is sometimes referred to as *aggregation* and simply declares an association between the client and server object. Support for these mechanisms is language-dependent. For example, C++ supports both composition and aggregation, where others such as Eiffel and Java only support aggregation.

Attribute objects can either be used as an alternative to implementation reuse through inheritance or in collusion with inheritance. When used as an alternative one benefits from the decoupling of the implementation of the used (server) object and the implementation of the client object. The host object can then delegate method responsibility to an attribute object. This highlights a tension between the development of a system through a natural model of the problem, and providing a model which can be implemented in the most efficient manner. For example, if an item **A** *'is a'* **B** with some added features, then the most natural object-oriented design is one where class **A** inherits from class **B**. However, a containment (or *'has a'*) relationship may be more appropriate as an implementation mechanism.

Development through attribution increases the potential for reuse. In a statically-typed language, any inheritance-based development must be declared at compile time. In contrast, if an object of class **A** 'has an' attribute of class **B**, the object assigned to the attribute variable can be of class **B** or any of its subclasses. This will be determined at run time. That is, attribution and inheritance can be used in tandem to deliver a design that maximises reuse.

In summary, OO provides several mechanisms for building software which take advantage of the primary rôle played by objects, and in most cases, their classes. One is also provided with a means of maximising the use of language constructs through polymorphism.

2.1.3 Polymorphism

Object-orientation supports three forms of polymorphism. The first is where one can send the same message to a collection of objects of different classes and each object will respond in an object-dependent way. Cardelli and Wegner [25] extending the polymorphism categorizations of Strachey [131] describe this form of polymorphism as *inclusion polymorphism* [25]. Together with *parametric polymorphism*, where a method or function works uniformly on a range of types, they comprise the two major forms of *universal polymorphism*. Although parametric polymorphism is universally supported by functional programming languages, it is only provided by a subset of OO languages. Eiffel's *generic classes* and C++'s *templates* allow classes to be declared with formal parameters, which are used to create instantiable classes when provided with an actual parameter [133, 84]. Thus one has the ability to achieve reuse over several types in a manner which is orthogonal to reuse via inheritance. Eiffel also provides *constrained genericity* where one can require the actual parameter to be of a particular class or one of its descendants, and thus guarantee a particular behavioural requirement of the generic class. We discuss (constrained and unconstrained) genericity further in Chapter 3.1, when comparing these approaches to functional programming's constrained polymorphism and parametric polymorphism.

The final form of polymorphism is *ad-hoc polymorphism*, where a method works (or appears to) on several different types in possibly different ways, and is often known simply as function/method identifier *overloading*. Ad-hoc polymorphism is in fact also supported in some non-OO languages.

Inclusion polymorphism is the dominant form of polymorphism within OO, whereas in functional programming parametric polymorphism is the dominant form and overloading is variably supported. This has a significant effect on the way one builds systems within the two paradigms. The OO approach is to factor out the common behaviour exhibited in various abstractions, and to build classes that support this behaviour. These are then the building blocks from which one can develop new abstractions with additional behaviour either through inheritance, composition or aggregation.

In the functional programming paradigm, one analyses the behaviour of functions. If more than one function exhibits the same behaviour it could be replaced by a single (polymorphic) function. In addition, if several functions have common patterns of computation they could be replaced by a single (higher-order) function.

An OO polymorphic variable can contain (or refer to) an object of more than one class. With statically-typed languages where each variable is declared with an explicit class, the contents of a polymorphic variable are constrained by the inheritance hierarchy. That is, the object must be of the declared class or one of its subclasses. In dynamically-typed languages all variables are polymorphic, since they can hold any value. Therefore all methods which take arguments are also polymorphic.

Any object that receives a message should be able to respond appropriately. That is, each object should deliver some common behaviour specified in its interface. If an object of class **X** supports the behaviour of objects of class **Y**, **X** is called a *subtype* of **Y** and **Y** a supertype of **X** [79]. Each object of a subtype can be used in place of an object of a supertype. A subtype is not necessarily a subclass and vice versa. For example, a subclass with less behaviour than its superclass is not a subtype. A subtype which is not related to its supertype through inheritance is not a subclass. However subtyping is typically introduced through inheritance. The main problem with achieving 'polymorphism through inheritance' is that inheritance is concerned with implementations, where subtypes focus on interfaces. That is inheritance supports code reuse by the 'implementor', where subtyping supports code reuse by 'clients' [108].

Java supports 'polymorphism without inheritance' by using a construct called an `interface` which provides a behavioural protocol, but no implementation. It is therefore similar to an abstract class, except that unlike an abstract class, one cannot provide any implementations for any methods of the `interface`. One is then able to achieve subtyping through an `interface` instantiation, since every class that implements the `interface` will be a subtype of the type specified by the `interface`.

Development Principles and Complexity

Although OO is often described as a natural way to develop software through its support for modelling the problem, developing an efficient, implementable solution is not a trivial task. This is signalled by the various laws, principles, and heuristics which guide the OO developer [114, 49, 85, 86]. Gamma et al [49] begin their book with the warning:

> Designing object-oriented software is hard, and designing *reusable* object-oriented software is even harder...Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimize it.

Meyer [84] argues that one should adopt the *open-closed* principle which requires software entities to be open for extension but closed for modification. That is, if one wants to add behaviour to a module then extend it do not change it. If one wants to increase the range of objects over which a function applies, then introduce a new class with the required behaviour. Satisfying this principle and many other principles comes at a cost and is not achieved by simply translating a 'natural' model of a problem into a design and then implementation. For example, if one needs to add behaviour to a parent class that is not currently supported by any of its child classes, one could extend the existing base class but would then need to restructure the class hierarchy. Thus although it has been argued that object-oriented software is easier to maintain than its alternatives [84], there is evidence to suggest that it often requires significant redesign and possibly even automated support [93].

The problems described above can be categorised as *same-paradigm problems* or the *essential complexity* between a design and implementation [20]. However, the *accidental complexity* which arises when one mixes paradigms is far more severe and difficult to resolve. As an illustrative example of this we describe the approaches to implementing in an OO language, a design that uses higher-order functions. With pure object-oriented languages one has to mimic 'functions as arguments' by applying a method to a parameter object whose only responsibility is a single method. That is, one needs to construct a stateless object whose only purpose is to support some behaviour. Since this *method object* or *function object* will act on the state of another object one breaks the encapsulation required of any OO model. In C++ one can overload the parenthesis operator `()`, which enables an object to be used as a function. C++ also supports a non-OO approach through the creation of a parameterised class, which can be instantiated with a pointer to a function.

In conclusion, OO models are best implemented in OO languages. In the following section we provide a brief overview of some modern OO languages.

2.2 OO Languages

In this section we present a brief overview of OO languages, highlighting their similarities and describing some of their differences. Object-oriented languages naturally support the features of object-orientation described in the previous section. This does not imply that every object-oriented model built using these features can be implemented in every object-oriented language, or that if they can they will result in the most effective and efficient implementation. However, it is more natural to implement an OO design in an OO language than in a non-OO language, because the development approach is the same. That is, they share common building blocks and glue, and a common development philosophy. We believe that this equally applies to any paradigm.

2.2.1 What is an OO language?

There are many descriptions of object-oriented programming or the properties required of an object-oriented language [71, 132, 87, 43]. The features possessed by languages that claim to be object-oriented include the ability to declare abstractions which support encapsulation and are extendible through inheritance, subtyping, and the binding of a method to a message at runtime (*dynamic binding*). Each OO language is either pure and sits wholly within OO, or includes features of other paradigms and is thus impure. Smalltalk, Java (which does however have non-object primitive types [3]) and Eiffel [115] are pure languages, where C++, Object Pascal [17], UFO (United Functions and Objects) [125] and OCaml (Objective Caml) [75] include various impurities. Further examples of impure OO languages are Pizza [105], which has added support for higher-order functions and parametric polymorphism to Java, and O'Haskell [91], an object-oriented extension of Haskell.

The typing mechanism of a language influences the scope of objects to which a message can be passed. Although statically-typed OO languages provide the benefits of compile-time type checking they also constrain the classes whose objects may receive a message. Statically-typed languages partially resolve this dilemma by supporting inclusion polymorphism through inheritance. Smalltalk, which is dynamically-typed or class-focused rather than type-focused [49], adopts the opposite approach of not catching any type errors at compile time, but having the freedom to send a message to any object that supports the appropriate behaviour through a matching method. Thus Smalltalk classes are not used for checking the type correctness of a program but to specify, and in most cases, implement objects. Most languages that are class-focused have classes as first-class citizens which can be manipulated at run-time. OO languages can be differentiated both through the type of inheritance they support and their approach to encapsulation.

2.2.2 Encapsulation and Inheritance

The interface provided by any object is dependent on the client object. In most OO languages, if the communicating objects are of the same class, then the interface is total or includes everything declared in the class. If the client object is of a subclass of the other object, then it is presented with an interface that includes all non-*private* entities. If there is no inheritance association between the objects, then the client object is presented with the most restrictive interface that only includes *public* entities. In contrast, Smalltalk, restricts access to an object's private parts to the object itself. That is, Smalltalk is truly *object-oriented* where each object fully encapsulates its state. Smalltalk enforces the encapsulation of state by making every attribute private and conversely, every method public. One is unable therefore to provide non-interface methods which are used to service interface methods. Other OO languages are not as draconian as Smalltalk, and allow the developer to decide on the (public and protected) interface of an object.

Many OO designs include classes that inherit features from more than one parent class through *multiple inheritance*. Although many problems are most naturally designed using multiple inheritance, it is not typically supported by OO languages. There are many reasons for this including the potential for ambiguity when invoking methods in response to a message. For example, when a message is passed to a child class object whose class doesn't provide an implementation of the required method, the message is deferred to a parent class (analogous to delegation in prototype/delegation languages). If both parent classes provide their own implementations the compiler will be unable to decide which one to invoke.

The large number of modern OO languages are matched by an ever increasing number of OO analysis and design methodologies. They can equally be categorised through their purity or impurity, and also through the approach to abstraction discovery.

2.3 OO Analysis and Design Methodologies

This section presents an overview of OO analysis and design methodologies (OOADMs). This is in no sense a complete overview. However it provides an insight into the essential features common to methodologies within the paradigm and how they support the development of OO software. A detailed, albeit dated critique is presented in [89]. A more recent survey of structured and OO techniques and methods is presented in [147], and a web-based comparative review of OOADMs can be found at [32]. In common with the imperative/structured paradigm, the OO paradigm supports the efficient and effective development of software. This is achieved by using development methodologies and implementation languages that use the same building blocks and glue. The methodologies are normally marketed through CASE tools that support their particular notation and techniques. The OO paradigm has a large set of such methodologies including the Booch Method [15, 82], OMT [120], BON [143], OOA/OOD [29, 28], Fusion [30, 80] and OOSE/Objectory [64]. Recently there has been a focus on developing a unified language, the Unified Modelling Language (UML) [16]. Although it is only a modelling language, and is therefore process independent

> it should be used in a process that is use case driven, architecture-centric, iterative, and incremental. [16]

The Unified Software Development Process has recently been developed using UML as its modelling language [63].

Each OOADM is a combination of a modelling language and a collection of integrated techniques which convert the results of requirements engineering into an implementable design. Most OOADMs provide a collection of diagrams that graphically represent various views of the models in development. Typically these diagrams can be used through all phases of development. Each OOADM can be categorised as pure, if it only models systems through communicating objects or their classes, or impure (or even hybrid) if action-oriented or data-oriented techniques and models are supported.

2.3.1 Pure and Impure OOADMs

A pure OOADM only uses object-oriented techniques and models to analyse a problem and design a solution. That is, the techniques aim to build models using objects and/or classes and their various associations. Functionality is analysed and described through communicating objects, and data is similarly described through its host objects. Impure methodologies combine object-oriented and non-object-oriented techniques, such as data-flow diagrams, into a single methodology. Examples include OMT [120] and the Schlaer and Mellor [126] approach which use functional models described through data-flow diagrams, and structural models using ERD type diagrams.

The Booch Method, OOSE, and BON are all pure object-oriented methods. For example, the Booch Method presents a static view of a system through *class diagrams*, a functional view through *object-scenario diagrams/object-interaction diagrams* and a dynamic view of the internals of an object via *state diagrams*. BON simply has *static diagrams*, *dynamic diagrams* and diagrams that present class details in a similar fashion to CRC (Class,Responsibility,Collaboration) cards [8]. CRC cards are used in many methodologies to record the name of a class, the attributes and methods it supports, and the other classes it collaborates with to achieve required functionality. They have typically been used in brainstorming sessions and can be physically arranged to illustrate particular designs.

Every methodology, pure and impure, supports the building blocks and glue of the OO paradigm. However, each methodology is typically described using its own modelling language and graphical representation of OO constructs and relationships. In the following section we present the steps in OO software development typically supported by an OOADM.

2.3.2 OO Development

OOADMs can be further classified by the driving factor of initial development. The classifications are *user-driven*, *data-driven* and *responsibility-driven*. With user-driven development the needs of the system users drive development. Jacobson [64] introduced *use case analysis* in his OOSE/Objectory method. Initial development models user interactions with the system through applications of use case analysis. We describe use case analysis later in this section. All OOADMs encourage an iterative approach to development. A system can initially be developed using a subset of user requirements, with any additional requirements introduced iteratively.

Data-driven methodologies such as OMT, initially focus on the major nouns in the requirements documentation and return a collection of matching objects and/or classes. The Booch Method and Martin and Odell's OOAD method [81] adopt a behaviour-driven approach, where the verbs in the requirements documentation guide the development of objects to support the behaviour indicated by a verb. Whichever approach is adopted there is a common underlying theme to development, which is summarized in the following list.

- Discovery of an initial collection of classes;
- description of class collaborations required to satisfy the system's functional requirements;
- assignment of responsibilities to each class;
- analysis of classes with significant state dynamics;
- development of classes, class collaborations and class responsibilities using behaviour scenarios;
- conversion of an analytical model which represents the problem to a design model of an implementable solution. New classes are introduced either to manage other classes or to reduce the coupling between existing classes.

Each methodology has its own techniques, notation and development themes. For example, OMT divides development into three modelling strands *object modelling*, *dynamic modelling* and *functional modelling*, OOA/OOD has the multilayer, multicomponent model, and BON encourages the development of models built on *seamlessness*, *reversibility* and *contracting*.

The initial step in object-oriented development is eliciting objects and their classes, from the deliverables of system's requirements engineering. The route to their discovery will depend on whether the methodology is user-driven, data-driven or behaviour-driven.

In each case, any data or behaviour are described through a host object. The development is immediately modelled through abstractions which encapsulate their state and host the methods which may act on the state. These abstractions should be extensible and each should model a real world entity or behavioural characteristic of the problem. Future development, for example of system functionality, is modelled through these abstractions. That is, functions or methods must be developed through communicating objects and guided by the interfaces of the objects.

We will illustrate the user-driven approach with a brief description of use case analysis [64]. A *use case* is a description of a set of sequences of actions that a system performs to achieve a desired result. Each sequence of events represents an interaction between system users, sometimes referred to as *actors*, and the system.

A use case is an analysis technique in that it captures the intended behaviour of the system, but does not specify how this is achieved. Each use case will be described by one or more scenarios which specify the semantics of the use case. For example a use case could be *'The data entry clerk inputs a result into a football results processing system'*. The textual description of the use case that includes details of the user will be translated into a set of scenarios which describe its achievement within the system. This may result in the introduction of new classes, new responsibilities assigned to existing classes and new collaborations between classes.

Wirfs-Brock et al. [149] subdivide OO software development into three phases: *initial exploration*, *detailed analysis* and *building subsystems*. The second phase puts the meat on the bones of the entities delivered by the first phase. One has to clarify through detailed inspection the class responsibilities - the data and methods - and the collaborations - inter-class dependencies - required of the system. Since classes are extendible one is encouraged to minimise the characteristics, both attributional and behavioural, of any class, and use inheritance and composition as mechanisms for building more complex abstractions.

Each class's attributes, methods, and collaborations can be gleaned from requirements information through the application of various analytical techniques including *use case analysis* and *CRC cards*. As one moves through analysis and into design a class's responsibilities and collaborations are scrutinized so that each class has a clear purpose and a high degree of cohesion, reuse is maximised, and inter-class dependency is

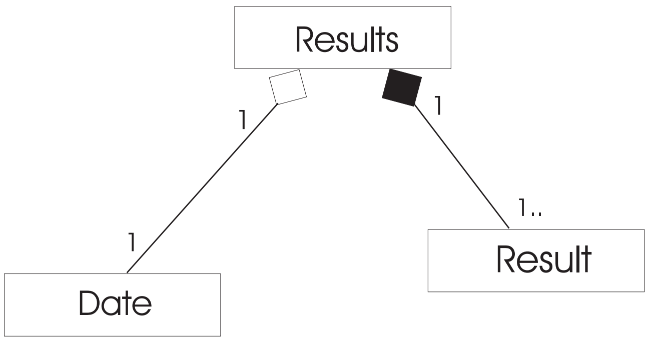

Figure 2: Class Diagram

minimised. One can use operational rules such as those described in the Law of Demeter [78] during such development.

Class collaborations are typically represented in *class diagrams*. These are similar to SSADM's *logical data structures* or entity relationship diagrams [26] in that they describe the major data entities in the system. In a class diagram the entities also include behavioural responsibilities. Figure 2 presents a class diagram where the class `Results` has a single `Date` attribute through aggregation, and one or more `Result` attributes through composition.

All OOADMs have a graphical notation for objects and classes, and their various collaborations. They also tend to support annotations which increase the semantics of the modelling languages. Thus one can present multiplicity of collaborations, or an insight into the actual relationship through textual information juxtaposed with the graphical notation.

A system's functional requirements are delivered through communicating objects. Since objects encapsulate state and behaviour, method development relies on the interaction of objects through message passing. Thus the appropriate metaphor is of a network of abstractions sending messages to other abstractions. In each case the receiving abstraction is responsible for managing the response to a message. One can present a view of function or method development through *object diagrams*. UML supports two types of object or *interaction diagrams*. *Collaboration diagrams* (object-scenario diagrams in the Booch Method, instance diagrams in OMT) have objects as the main subjects, and methods are described through messages passing between the objects. The

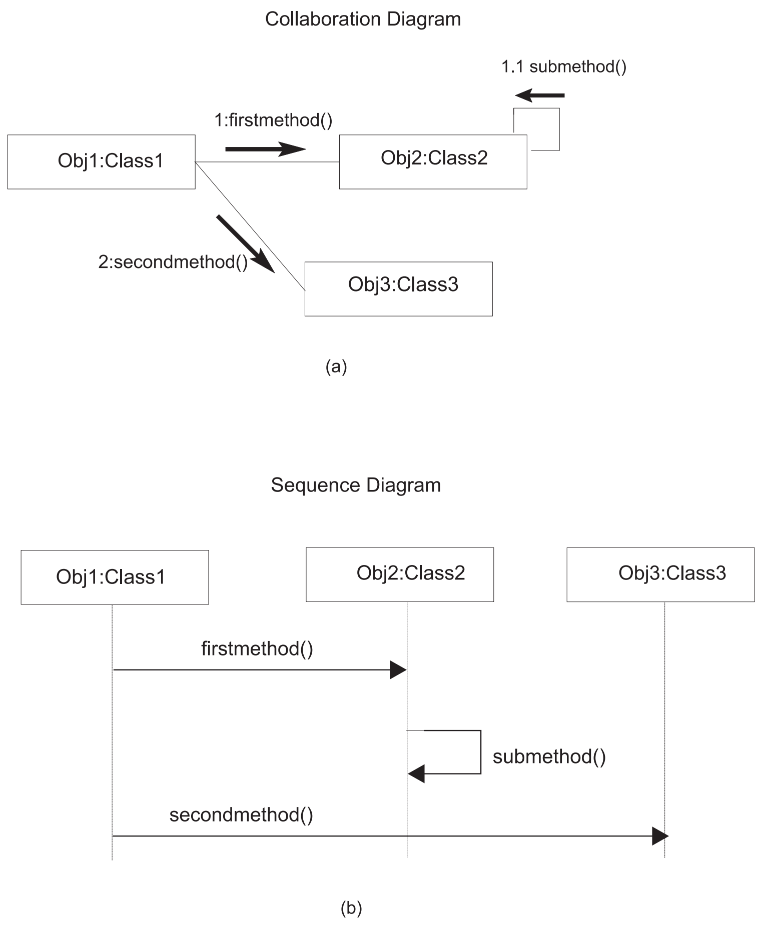

Figure 3: Collaboration and Sequence Diagrams

reverse is the case with *sequence diagrams* in which the messages take pre-eminence over their associated objects. Sequence diagrams emphasise the ordering of the messages, where the emphasis in a collaboration diagram is on the objects that are communicating. Sequence diagrams have similarities to Gantt charts (a popular graphical representation of a project's activities) of use within a field of operations research. We give examples of these types of diagrams in Figures 3(a) and 3(b).

The models described thusfar focus on the static, and functional requirements of a system. Many objects' response to a message will be state-dependent. That is, the values of an object's attributes will often influence an object's behaviour. One can model these object state dynamics through *state diagrams* which describe a collection of object states, and the actions which lead to transitions between the states. The state diagrams used by most OOADMs follow the notation of Harel [54].

Once an acceptable analytical model of the problem is in place, the focus turns to the development of an implementable design. At this stage one may introduce classes that manage the interaction of other classes, or others which support some common behaviour required of existing classes. Where analytical models simply reflect a system's requirements, design models need to be efficient, effective and implementable. The developer can adopt various principles, laws, and existing designs during this process.

2.4 Summary

In conclusion, the object-oriented paradigm has marketed itself as a packaged development approach. From the initial stages of development one can describe a problem in terms of OO elements and models using one of a significant number of modelling languages, methodologies and supporting CASE tools. The translation from an abstract model to implementation code is eased through removing the accidental complexity incurred when switching paradigms. Although translating a model of the problem into an effective and efficient model of a solution is not a trivial task, once achieved there are a large number of OO programming languages in which OO models can be naturally implemented.

The functional programming paradigm is currently without any analysis and design methodologies. Therefore, if one wants to model a problem one either has to adopt an ad hoc approach or use an existing non-functional methodology. In the following chapter we describe the functional programming paradigm, placing emphasis on the features which have a major influence on the design of functional software.

Chapter 3

Functional Programming

In this chapter we clarify the main features of the functional programming paradigm and how they influence software development within the paradigm. We begin with a brief overview of the paradigm that lists its major features. These are: functions as the basic unit of program development; strong typing as an aid to development pre-implementation, during implementation and post-implementation; parametric polymorphism and the first-class nature of functions as the major routes to reuse; and, the support provided for developing user-defined datatypes. Each of these features are described with illustrative examples, and, where appropriate we draw comparisons with approaches adopted within the OO paradigm. For example, parametric polymorphism is supported by both paradigms, but has a greater influence on software development within the functional paradigm. In Section 3.2 we review features which are either variably supported or are supported by a significant minority of modern functional programming languages (FPLs). These include lazy evaluation that supports programming with infinite data structures, overloading of function names, and the mechanisms for delivering modularity-in-the-large. We include various pointers to the modelling of functional designs using FAD. For example, we introduce the FAD units *exclusive signature* and *permissive signature*. These are defined briefly in this chapter, with a more detailed coverage provided in Chapter 5. In the final section, we present the arguments for the need for (and requirements of) a functional analysis and design methodology (FADM). Chapter 4 provides a more detailed argument in support of analysis and design methodologies.

3.1 The Functional Programming Paradigm

The functional programming paradigm, in its purest form, is about building programs from functions. Each function computes a value that depends solely on the values of the function's inputs. Every function has a type that in most functional languages is determined statically, and functions are first-class and thus can be treated as data values. If an OO system is built through

> a collection of interacting abstractions that host their own state, provide mechanisms for manipulating the state, and deliver an explicit behavioural contract to other abstractions

functional programming relies on

> a collection of abstractions that generate values dependent only on the values they receive.

Functional languages also offer significant support for modular development and thus for programming-in-the-large. Although it is beyond the scope of this thesis to provide an exhaustive list of features of the paradigm, we list below those features which we believe have the most significant impact on how one develops software within the paradigm. The following subsections present details on each feature in turn with some commentary on its influence on the development of software within the paradigm. We will illustrate many of the features with example code written in Hugs 98 [70]. The functional programming paradigm includes the following features:

- functions as the fundamental building-blocks of programs;
- strong typing;
- parametric polymorphism;
- functions as 'first-class citizens'; and,
- substantial support for the development of user-defined types, both concrete and abstract.

Collectively these features describe a clean, mathematically tractable and robust technology with significant support for reuse. It enables the developer to focus directly on the functional characteristics of a system without either the loss of data security inherent in imperative programming, or the indirect approach imposed by the object-oriented paradigm.

3.1.1 Functions, Values and Referential Transparency

Programming in a functional style using a functional language involves building definitions and evaluating expressions. As Bird and Wadler [11] concisely state:

> The primary role of the programmer is to construct a function to solve a given problem.

The behaviour of these functions depend only on the values of their arguments, and not on the value of any variables which model the state. Thus functional programming encourages a view of computing that is significantly different to that of a sequence of state modifications.

Imperative programs are built through a collection of mutable variables which model the state, and procedures which modify these (typically global) variables. The behaviour of the procedures typically depend on the values of the mutable variables, which can be changed as the procedures run. There are various problems with this approach. Global data is inherently insecure since there is no explicit restriction of access to a variable's contents, and it can be difficult to understand a program given that the value contained by any variable will depend on the program itself. Non-modular, unstructured programs written in an imperative style also suffer from multiple entry and exit points and little support for programming-in-the-large [38]. Although structured programming [35] has addressed some of these problems, and object-based languages such as Modula-2 [146] have addressed the issue of modular software development, the imperative paradigm has generally lacked significant modular support.

Object-orientation has addressed these issues through the encapsulation of variables with the procedures that act on the variables within abstractions called objects. Although the variables remain mutable, and thus their contents can be changed, access to a variable is constrained by the interface supplied by the object that hosts the variable. Objects, and not variables and independent procedures, are the units upon which a program is developed. New functionality is developed through collaborating objects rather than directly gluing together existing procedures.

The (pure) functional programming paradigm has adopted a quite different approach. All variables are immutable. That is, variables in the functional programming context (in common with mathematics) do not vary but always denote the same value. Functions are therefore the mechanisms for creating new values and not for updating the values of existing variables. That is, a function takes one or more input values and returns a new value that is determined completely by the inputted values.

This has a significant impact not only on how one builds a program, but also on the meaning of a program. The meaning of an imperative or OO program is understood by the effect it has on the state (the collection of variables) of the machine as it runs. In contrast, the meaning of a functional program is understood by the values it computes. That is, the meaning of an expression in a pure functional language is simply its value. There are no *side effects* (state changing actions) accompanying the evaluation of an expression.

One benefit of using a side effect-free language is that any expression of the language that has a well-defined value can be evaluated in any order. Order of evaluation only matters when a variable's value may depend on the order of evaluation of some subexpressions. Many pure functional languages can therefore support non-strict semantics whose influence on software development we describe in Section 3.2.1.

An expression written in a side effect-free language can have any subexpression substituted by its value without altering the value of the expression. This characteristic is a particular case of *referential transparency*, the ability to substitute equals for equals. Since an expression 'equals' its value the substitution will not affect the value of the expression.

In conclusion, in a pure functional language all computations are performed via function application. Ingenious mechanisms for supporting impure interactions such as I/O have been developed, the most recent of which is the monadic approach adopted by Haskell [53, 103]. Software development within the functional programming paradigm is built predominantly on functions. Various mechanisms exist for building new functions from existing functions and maximising the scope of a given function, some of which are described in the following sections. The scope of a given function is intimately tied to its *type*. We describe in the following section how a function's type constrains the application of a function, and in Section 3.1.3, how parametric polymorphism allows the functional programmer to reuse a single function over more than one type.

3.1.2 Strong Typing

Most modern functional languages are *strongly typed*. That is, every well-formed expression of a functional language has a type that can be determined at compile time. This means that no run-time errors are due to type mismatches. Just as the value of an expression depends only on the values of its subexpressions, the type of an expression can be deduced from the type of its components' expressions. For example, the function `frontPlusBack` is defined as follows:

```
frontPlusBack x = head x + last x
```

From the right hand side of the definition we can determine that the function is well defined if `x` is a value of any list type (denoted `[a]`), since the functions `head` and `last` take values of any list type and return the first and last element of the list respectively. In addition, since the values returned by these functions are added together, the list must contain numeric values. In Haskell we write that `frontPlusBack` has the type

```
Num a => [a] -> a
```

where `a` is a *type variable*, and `Num a =>` constrains the binding of the type variable to numeric types.

Since strong type checking involves type inference, the developer is not required (but is encouraged) to specify the type associated with each definition.

Therefore, the function `frontPlusBack` should be defined with an accompanying *type specification*.

```
frontPlusBack :: Num a => [a] -> a
frontPlusBack x = head x + last x
```

An explicit type specification is encouraged since it aids software development in several ways which include:

- a signal to the type-checker regarding the expected type of the associated entity;

- a guide to the requirements of a function in terms of its expected input and required output. This can be used both in advance of implementation of the entity and as an interface to entity use;
- a documentation device; and,
- as a pointer to potential reuse of library constructs where functions can be categorised by their types.

Strong typing therefore provides support both at the implementation stage of development and during pre-implementation analysis and design. The type of a function is a constraint on how the function can be used. This could lead to a rather inefficient and expensive approach to development, where functions have to be redefined every time one wants to use them over a different type. However, in common with statically-typed, object-oriented languages, mechanisms exist for minimising this cost and maximising the scope of use of existing entities. Where statically-typed, class-based, object-oriented languages have adopted *inclusion polymorphism* as the predominant mechanism for reuse, functional languages support *parametric polymorphism*.

3.1.3 Parametric Polymorphism

In Chapter 2 we described how inclusion polymorphism is the dominant form of polymorphism supported by object-orientation. Inclusion polymorphism supports the notion of 'one type many methods' where the method called is determined dynamically through the class of the object that receives the message rather than the declared class. Parametric polymorphism can be viewed as the antithesis of inclusion polymorphism. Parametric polymorphism enables 'one function many types', where a function is not restricted to single monomorphic types but can be used over a range of types. However, the arguments of a polymorphic function must themselves be monomorphic. Polymorphic arguments require rank-2 polymorphism which although supported, for example, by Hugs 98 [70], is not a ubiquitous feature within the paradigm. Polymorphism in the functional world therefore supports the reuse of code rather than the ability to supply arguments of various forms with a common interface.

One can achieve significant reuse within the functional programming paradigm by taking advantage of parametric polymorphism. If two or more monomorphic functions with the same arity exhibit common behaviour over values of distinct types, they could possibly be replaced by a single polymorphic function. For example, the Haskell function `length` which takes a list of values and returns the number of items in the list, operates in a consistent fashion for a list of elements of any type. Similarly, the pair selector functions `fst` and `snd` require no specific characteristics of the pair element values, and thus can be applied to pairs whose elements are of any type.

```
length          :: [a] -> Int
length           = foldl' (\n _ -> n + 1) 0

fst            :: (a,b) -> a
fst (x,_)       = x

snd            :: (a,b) -> b
snd (_,y)       = y
```

In OO one could achieve a similar form of reuse through C++ templates or Eiffel's generic classes. For example, in C++ one can declare a parameterised container class `List<Type>` which includes a method which returns the length of a list. One can create instantiable classes by providing the parameterised class with an actual parameter such as `List<String>`, a list of strings class, and `List<Person>`, a list of people class. Since the method which returns the length of the list, and all other methods of the parameterised class, requires no particular characteristics of the actual parameter class, the same method can be applied over objects of any instantiating class. Some languages in both paradigms support constrained parameterisation in which the actual parameter is required to support some particular behaviour. This is described in Section 3.2.2.

An important indicator of potential parametric polymorphism is the lack of behaviour required over the types associated with a function or the types that provide the values for a data structure over which a function is defined. That is, although the function `length` requires the container type (in this case a list) to support certain behavioural requirements, the type that provides the values contained in the list has no such requirements. The function `length` can be applied to a list of any type, since it does not require a list's values to conform to any particular specification. This is also true of the pair selection functions.

Where parametric polymorphism supports the use of a single function over many types, *higher-order functions* which take functional arguments capture common patterns of computation between several functions. In the following section we describe the influence that 'functions as values' has on software design within the functional programming paradigm.

3.1.4 First-Class Citizens

Hughes [57] argues that the two features of functional languages which have the most significant impact on (small scale) modular development are *higher-order functions* which rely on the first-class citizenship of functions and laziness. Since laziness is not a feature of all functional languages it would be inappropriate to describe it as a feature of the paradigm. However it is supported by a significant minority of pure functional languages and we will describe it in Section 3.2.1.

One way of distinguishing the OO paradigm from the functional programming paradigm is through which constructs are *first-class*. Where objects are first-class citizens in an object-oriented language and thus can be treated as data, functions are first-class in functional programming. Therefore, a function can be an argument of a function, returned by a function, or an element of a data structure.

Functions that either take functions as arguments or return a function as a result are classified as *higher-order functions* or *functionals*. They provide a significant glue for building programs in the functional programming paradigm. Functions with multiple arguments can be modelled in a *curried* form where they take their arguments one at a time. The *uncurried* form typically presents the arguments in a tuple. Curried functions can be partially applied to return a new function. These functions can either be created at compile time or at run time. The functions `curriedPlus` and `uncurriedPlus` illustrate these two forms, and `plus5` is a function created through the partial application of the function `curriedPlus` to the argument `5`.

```
curriedPlus     :: Int -> Int -> Int
curriedPlus m n         =  m + n
```

```
uncurriedPlus   :: (Int,Int)  -> Int
uncurriedPlus (m,n)     =  m + n

plus5           :: Int -> Int
plus5                   =  curriedPlus 5
```

Functions that take functions as arguments model common patterns of computation between several first-order functions. For example, the functions `doubleList` and `trebleList` multiply every integer in a list by two and three respectively. They can be replaced by a single higher-order function `applyArithList` which takes an arithmetic function as its first argument, a list of integers as its second argument and returns the list where the function has been applied to each element.

```
applyArithList :: (Int -> Int) -> [Int] -> [Int]
applyArithList f ls = map f ls
```

Functionals are not unique to the functional programming paradigm but are implemented more naturally than in non-functional languages. For example, in C one can indirectly use functional arguments through pointers, and Pascal supports functional arguments of a simple kind but not functional results. In object-oriented languages where objects not functions are first-class, there are various mechanisms for mimicing functions as arguments. These include: applying methods to function objects (objects with no state and a single method); applying methods to objects with an overloaded parenthesis operator (in C++); taking advantage of impurities in certain languages and using templates/generics; and, by using static methods (in class-based languages) which can be called without reference to an object.

Hutton's paper *Higher-order functions for parsing* [59] presents a collection of higher-order functions (or combinators) which are used to build parsers through the combination of existing parsers. More recently Hutton and Meijer have re-implemented the combinators using monads [60] to which we will refer in Section 3.2.4. Through a collection of combinators such as `then`, `alt` and `using` which correspond to sequencing in BNF, alternation in BNF, and the $\{\ldots\}$ operator in Yacc, and a collection of primitive parsers which amongst other things represent success and failure, one can quickly build recursive descent parsers which are both simple to understand and easy to modify. This is not the case with parsers developed using imperative or OO languages.

We illustrate parser combinators below. The functions are written in Haskell rather than Miranda[1] as used in Hutton's paper.

```
type Parser a  = String -> [(a,String)]

alt :: Parser a -> Parser a -> Parser a
p1 ‘alt‘ p2  = \ inp -> p1 inp ++ p2 inp

then :: Parser a -> Parser b -> Parser (a,b)
p1 ‘then‘ p2 inp = \ inp -> [((v1,v2), out2)
                             | (v1, out1) <- p1 inp,
                               (v2, out2) <- p2 out1]

using  :: Parser a -> (a -> b) -> Parser b
p ‘using‘ f  = \ inp -> [(f v, out) | (v, out) <- p inp]
```

The first line of the code declares the type `Parser` which is parameterised over the type of result values. In Hutton's paper the parser type was parameterised over the input value type as well. A parser is a function that takes a collection of input tokens (as a string of characters) and returns a list of 'parsed input/unconsumed input' pairs as results. A list of results is returned so as to deal with an ambiguous underlying grammar.

The combinator approach to parser generation differs from that of parser generators such as Lex and Yacc [2] and Happy [50], in offering an extensible rather than a fixed set of combinators for describing grammars. Another example of a combinator approach to functional development is described by Wallace and Runciman [144] who have developed a toolkit of components for processing XML documents in Haskell which includes a set of combinators for scripting stylesheets and a set of selection combinators.

Any functional analysis and design methodology must both encourage and support the development of higher-order functions. FAD's modelling language includes a graphical representation for curried functions and supports function development through the

[1]Miranda is a trademark of Research Software Ltd

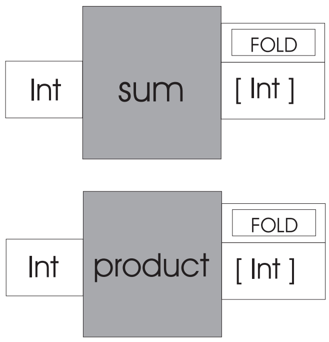

Figure 4: Higher-order Development

partial application of a function to an incomplete set of arguments (see Section 5.4.3). In addition, one is encouraged to use *permissive signatures* to discover higher-order functions. A permissive signature provides a specification of functions defined over an associated type. It does not provide an interface to a type, but rather a guarantee that the functions specified are defined over the type. Permissive signatures are fully described in Section 5.2.3, but we briefly illustrate their use with the functions `sum` and `product`. Each function takes a list of integers and return their sum and product respectively. Both functions require the elements in their argument lists to be combined using an arithmetic operator. That is, they both require that 'folding' behaviour be supported by the list type. One can make this common pattern of behaviour explicit through the association of a permissive signature - in this case `FOLD` - with the list type. This is illustrated by the FAD representations of the functions `sum` and `product` in Figure 4.

Although this does not guarantee that a higher-order function would be appropriate, it certainly signals that it is a possibility. See Chapter 5 for full details on FAD's modelling language and graphical notation, and Section 7.3.3 for a fuller description of this approach to the discovery of potential higher-order functions.

Thusfar the functional programming features described have been largely function-oriented. The last ubiquitous feature which we believe has a significant influence on software development is the support for user-defined types.

3.1.5 User-Defined Types

Every modern functional language provides a wealth of built-in types. These include base types such as the type of characters and the Boolean values, and various composite types such as tuple types and function types. However the languages in the paradigm also provide the developer with mechanisms for developing new types. The predominant mechanism is through the declaration of *algebraic types*.

Algebraic Types

Algebraic types are a single mechanism for the creation of various forms of types that would otherwise have to be delivered through separate mechanisms. These include, *sum types* which have alternative domains and *product types* which are types with multiple components. They are called 'algebraic' since they are examples of term (or initial) algebras whose elements are uniquely created through a set of *value constructors*. Here we must make a distinction between value constructors which construct values of a type, and *type constructors* which construct types. In Haskell, algebraic types are declared using the keyword `data`, and introduce a new type constructor such as `TC`, and one or more new value constructors, `VC1`, `VC2` and so on.

```
data TC tv1 ... tvk = VC1 t11 ... t1m  | ... | VCn tn1 ... tnp
```

A type constructor can take zero or more parameters made explicit by the type variable(s) which follow its name. We have represented these as `tv1`, `tv2` and so on. Each value constructor may take one or more parameters, which in each case will either a type variable used by the type constructor or a type. We have named these `t11` to `tnp`.

For example, the algebraic sum type `IntOrFloat`, in common with the built-in types `Int` and `Char`, is a *nullary type constructor* since it takes no parameters. Its values are constructed by applying the *unary value constructor* `ConsInt` to an `Int` value, or the *unary value constructor* `ConsFloat` to a `Float` value. `anIntValue` and `aFloatValue` are both values of type `IntOrFloat`.

```
data IntOrFloat = ConsInt Int | ConsFloat Float
anIntValue      = ConsInt 3
aFloatValue     = ConsFloat 3.0
```

```
data Days               = Sunday | Monday | Tuesday | Wednesday|
                          Thursday | Friday | Saturday

data ThisOrThat a b     = This a | That b

data Tree a             = EmptyTree | Node a (Tree a) (Tree a)
```

Figure 5: Algebraic Types

For the remainder of this thesis, to ease exposition, we will refer to nullary type constructors simply as types and non-nullary type constructors as type constructors.

The algebraic type mechanism also supports

- *enumerated types* through the declaration of a set of nullary value constructors. This is illustrated in Figure 5 with the type `Days`;

- parameterised types. These are types that are created through the application of a type constructor to one or more parameters. Each type constructor has a *kind* which specifies the number and form of parameters of the type constructor. That is, a kind is to type constructors what a type is to functions [66]. Using Jones' notation [66], all types have the kind `*`, and the kind $\kappa_1 \rightarrow \kappa_2$ represents type constructors that take an entity of kind κ_1 and returns one of kind κ_2. This is illustrated in Figure 5 with the parameterised type `ThisOrThat a b` whose type constructor `ThisOrThat` takes two parameters of any, and possibly differing, types. For example, values of the type `ThisOrThat Int Bool` are created through the application of the unary value constructors `This` and `That` to `Int` and `Bool` values respectively. The constructor `ThisOrThat` has kind `* -> * -> *`;

- *recursive types* which are described in terms of themselves. This is illustrated in Figure 5 with the type `Tree a`.

Functions over an algebraic type that have value-dependent behaviour are most naturally defined using *pattern matching*. Actual arguments are matched against an argument pattern presented in a function definition, and if successful the associated expression is evaluated. If the match fails, the next argument pattern is checked and so on. For example, the polymorphic function `zeroOrOne` takes a value of type `ThisOrThat` `a b` and returns `0` if the value is constructed using the value constructor `This` and `1` otherwise. That is:

```
zeroOrOne :: ThisOrThat a b -> Int
zeroOrOne (This _)      = 0
zeroOrOne (That _)      = 1
```

The underscore is the wildcard pattern that can be used when a part of a pattern is not used in the body of the function definition.

Clearly pattern-matching requires that the function has access to the implementation of the type, which results in tight coupling between the function and type. This is poor modular design, since any change in the type implementation will require a change to the function definition. A modular approach built on *information hiding* is achieved by using *abstract data types* which we describe in the following section.

Abstract Data Types

Abstract data types support a separation of a type's interface from its implementation. They are a mechanism for decoupling a type and its clients. An abstract data type is a type with an explicit collection of operations defined over the type. These operations are specified in the interface to the type. Thus one can only use values of the type by using one of its interface operations. One can then reimplement the type and its operations with the interface remaining consistent for any existing or future client.

Abstract data types are therefore integral to the modular development of functional programs. It is therefore essential that they are both supported in any functional modelling language, and play a predominant role in the methodology. The mechanism(s) for the implementation of abstract data types is language-specific. Many functional languages use *modules* as the type host, which is accompanied by a restrictive interface. We describe modules in functional languages in Section 5.3.1. We present here a brief overview of the various mechanisms for implementing abstract data types (ADTs).

Miranda uses the keyword `abstype` to declare such a type, which is followed by the type's identifier and interface, which is presented as a collection of type specifications. SML has both a keyword and a means of abstracting the contents of a structure through an opaque signature. Any types declared in such a structure will become abstract due to the associated signature. The signature provides full syntactic details regarding each of its entities. Haskell supports ADTs through its module system. A module export list that includes a type without its value constructors declares the type as abstract. However the type's operations are simply named without any type specification. Clean delivers ADTs through their definition modules, which are similar to SML's signatures except that each implementation module can be associated with only one definition module.

Abstract data types are essential to the development of a modular system, whose components can be modified, reused, and maintained, in an efficient and effective manner. FAD supports abstract data types through the assigning of a type to a module, and associating an *exclusive signature* with the module. An exclusive signature is a collection of entity specifications which, when associated with a module which hosts the entities, acts as an *interface* to the module. That is, a client of the module has access only to those entities specified in the exclusive signature which mediates use of the module by the client. One can impose abstraction on a type by hosting it in a module whose clients have no knowledge of how the type is constructed. That is, the type is specified in an exclusive signature **E** but its *constructor signature* is absent. A constructor signature is a permissive signature which specifies the value constructors of a type. This example highlights the differing roles of the two forms of signature provided by FAD. An exclusive signature provides an interface to a construct which hosts various declarations, whereas a permissive signature declares a minimal set of operations over one or more types.

A type is therefore not abstract by default, but instead can have abstraction imposed when used by an entity of another module.

The module `AbstractTypeModule` hosts the type `ThisOrThat a b` but only exports its type constructor and not its value constructors. Thus any entity of another module which uses the type, uses it as an abstract type via the operations specified in the export list that follows the module name in parentheses. In this example, the type constructor `ThisOrThat` is accompanied by two selection functions `get1` and `get2`. In FAD, the export list will be modelled as an exclusive signature.

```
module AbstractTypeModule (ThisOrThat, get1, get2) where
data ThisOrThat a b = This a | That b
```

```
get1 :: ThisOrThat a b -> a
get1 (This x) = x
get1 _        = error "Inappropriate application"
get2 :: ThisOrThat a b -> b
get2 (That x) = x
get2 _        = error "Inappropriate application"
```

The development of module (and subsystem) architectures and the development of associated exclusive signatures are integral to the FAD methodology. Full details of modules, exclusive signatures and abstract data type support are provided in Chapter 5. The methodology is described in Chapter 7.

In the following section we discuss features which are common to significant subsets of functional programming languages, but are also important in influencing the way one develops functional programming software. The section begins with a very brief scan of the differing characteristics of modern functional programming languages.

3.2 Other Features

Functional programming languages are characterised in various ways. For example, Haskell, Miranda and Gofer [65] are pure, non-strict, sequential languages. ML is an impure, strict, sequential language. Erlang and Clean [106] are concurrent languages that are impure, strict and pure, non-strict respectively. All of these languages deliver the functional programming features described in Section 3.1. Impure languages also support features typically associated with imperative languages such as variable assignment. FAD does not support impure features.

Although FAD describes software models which may be implemented using any functional language, a significant minority of functional programming languages support non-strict semantics through lazy evaluation, which encourages a particular approach to program design and development. The following section provides a review of *laziness* and its impact on software development.

3.2.1 Laziness

Programming languages are initially classified first through the (predominant) paradigm they support, and then by their type-checking approach. Functional languages are further classified as either *strict* or *non-strict*. Languages with strict semantics, supported by eager evaluation (or call-by-value reduction), force the full evaluation of all arguments. In contrast, those with non-strict semantics delivered through lazy evaluation (or call-by-need reduction), only require those arguments that are needed in the function body expression to be evaluated [97]. That is, every argument is evaluated exactly once in strict languages, and at most once in non-strict languages. When both approaches lead to termination the values returned will be identical. However there are simple examples that will not terminate when using eager evaluation, such as the application of the function `fst` - which selects the first element of a pair - to a pair whose second element is undefined.

```
fst           :: (a,b) -> a
fst (x,_)      = x

f       = fst (True, 1/0)
```

Since the function `fst` only uses the first element of a pair on the right hand side of the definition, the second element will not be evaluated when using lazy evaluation. Thus `f` will evaluate to `True`. With eager evaluation, both parts of the pair need to be evaluated, and hence, `f` would be undefined.

Lazy evaluation distinguishes most pure functional languages from imperative languages and most object-oriented languages. Programs written in those languages often rely on side effects, which are intimately linked to evaluation order and thus require strict semantics where evaluation order is clear. Lazy evaluation is effectively 'demand driven evaluation', and hence it is more difficult to predict evaluation order, and therefore harder to predict when side effects will take place.

Lazy evaluation supports programming with infinite data structures, such as infinite lists, through enabling partial evaluation of a data structure. For example, the higher-order function `filter` takes a predicate and a list and returns those elements of the (possibly infinite) list that satisfy the predicate. The higher-order function `take` takes an integer `n` and a list and returns the first `n` elements of the list. If `filter even` is composed (denoted `.`) with `take 2`, the resulting function will return the first 2 even numbers in a list.

```
first2Even :: [Int] -> [Int]
first2Even xs = (take 2 . filter even)
```

With lazy evaluation one only evaluates as much of the list as is required to return 2 even numbers. Thus as each even number is confirmed, it is outputted until 2 numbers are returned. That is, if we apply `first2Even` to the infinite list of positive integers, denoted `[1..]`, evaluation proceeds as follows where '$\leadsto$' indicates a step of the calculation.

```
first2Even [1..]
                  ⇝  (take 2 .  filter even) [1..]
                  ⇝  (take 2 .  filter even) [2..]
                  ⇝  2 :  (take 1 .  filter even) [3..]
                  ⇝  2 :  (take 1 .  filter even) [4..]
                  ⇝  2 :  4 :  (take 0 .  filter even) [5..]
                  ⇝  2 :  4 :  []
                  ⇝  [2,4]
```

Laziness has enabled a modular design where there is a separation of value generation and value use. One is therefore able to adopt a software development approach where behavioural requirements are delivered by separate entities that can be independently developed and maintained.

3.2.2 Overloading

All modern functional languages support parametric polymorphism. However recent developments within several languages deliver support for the middle ground between monomorphism and polymorphism. The motivation for this development is that there are many examples where monomorphism is too restrictive and polymorphism is too general. For example, the function `sum` of Section 3.1.4 could be given a monomorphic type

```
sum :: [Int] -> Int
```

which disallows application to lists of other numeric values. Alternatively we could give it the type

```
sum :: [a] -> a
```

that suggests that the function can be applied to a list of non-numeric values, which is clearly not the case.

The OO language Eiffel provides *constrained genericity* to solve this problem of *constrained parametric polymorphism* [84]. Whereas unconstrained genericity allows any actual parameter to be bound to the formal parameter of a generic class declaration, constrained genericity requires the parameter to be of a stated class or one of its subclasses. Thus one can guarantee that a required behaviour is supported by any potential instance object.

A collection of functional languages, such as Haskell and Clean, have resolved this dilemma through supporting constrained polymorphism via *type* and *constructor* classes. A type class is a collection of types. Type constructors of the same *kind* can be collected in constructor classes. Current language support is largely restricted to single parameter classes, multiple parameter classes which collect associated type constructors of the appropriate kinds are supported, for example, by Hugs98.

Each type or constructor class specifies a collection of entities with their type specifications. A type or type constructor instantiates a class when each specified entity is matched by one of the same name defined over the type constructor, and with a type that is an instance of that specified in the class. Thus one may *overload* function and value names in a controlled manner using this mechanism.

The type class `ZeroOne` specifies the function `zeroOne` which takes a value of an instantiating type and returns either `0` or `1`. The constructor class `EmptyOrNot` specifies the function `emptyOrNot`, which takes a value of an algebraic type whose type constructor has the kind `* -> *` and returns `0` if it is ‘empty’ and `1` otherwise. The types `Int` and `Bool` instantiate the type class `ZeroOne`, and the type constructors `[]` (the list type constructor), and `Tree` instantiate the constructor class `EmptyOrNot`.

```
class ZeroOne t where
        zeroOne :: t -> Int
instance ZeroOne Int where
```

```
        zeroOne i
           | even i     = 0
           | otherwise  = 1
instance ZeroOne Bool where
        zeroOne False   = 0
        zeroOne _       = 1
class EmptyOrNot c where
        emptyOrNot :: c a -> Int
instance EmptyOrNot [] where
        emptyOrNot []   = 0
        emptyOrNot _    = 1
instance EmptyOrNot Tree where
        emptyOrNot EmptyTree = 0
        emptyOrNot _         = 1
```

In common with types, the languages that support type classes provide built-in classes and enable the user to define new classes, extend existing classes, or instantiate existing classes. A type/constructor class presents an interface that is implemented by any type/type constructor that instantiates the class. For example, all numerical types instantiate the (single parameter) type class `Num`'s interface that includes various arithmetic operations and numeric functions. We present the class in an elided form below followed by a collection of instantiations.

```
class (Eval a, Show a, Eq a) => Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a

-- instances:
instance Num Int
instance Num Integer
instance Num Float
```

```
instance Num Double
```

The class `Num` extends the interfaces of the classes `Eval`, `Show` and `Eq`, and is instantiated by the types `Int`, `Integer`, `Float`, and `Double`. We can now declare `sum` as follows:

```
sum :: Num a => [a] -> a
sum = foldl (+) 0
```

where the context `Num a =>` states that the type variable `a` is constrained to range over types that belong to the type class `Num`. `sum` can be applied to a list of values of a type in the type class `Num`. The version of the addition operator used is determined by the type of values in the list.

Constructor classes support *higher-order polymorphism* or the application of functions uniformly over (potentially) all type constructors of a particular kind [66]. For example, the constructor class `Fold` specifies folding behaviour through a collection of functions. The class has a single parameter of kind `* -> *`, and thus can be instantiated by unary type constructors such as the list constructor `[]`.

```
class Fold f where
        ffoldl          :: (a -> b -> a) -> a -> f b -> a
        ffoldl1         :: (a -> a -> a) -> f a -> a
        ffoldr          :: (a -> b -> b) -> b -> f a -> b
        ffoldr1         :: (a -> a -> a) -> f a -> a
```

Now we can declare a function `sumC` that sums the numeric values contained in any data structures built using an instantiating type constructor. The version of `ffoldl` used depends on the argument type of the function.

```
sumC :: (Fold c, Num a) => c a -> a
sumC = ffoldl (+) 0
```

In conclusion, type/constructor classes deliver a methodical approach to function name overloading. They provide a mechanism for associating a collection of types or type constructors that support some specified behaviour, which is typically indicated by the name of the class. We can regard constrained polymorphism as a natural generalisation of polymorphism, where polymorphism is simply unconstrained use of the general case. That is, where polymorphism delivers abstraction over any type, constrained polymorphism requires the types to support some specified behaviour.

FAD represents type and constructor classes through *permissive signatures*. However, a permissive signature does not have to be implemented as a type or constructor class. Permissive signatures indicate that a type must support some stated behaviour or that a function requires a certain behaviour over one of its types. Whether the implementation involves overloaded functions and type classes will depend both on the implementation language and other design decisions. A full description of FAD's permissive signatures is presented in Section 5.2.3, and the development of permissive signatures to support functions and type development is described in Section 7.3.1.

3.2.3 Modular Development

Modern functional languages, in common with their object-oriented counterparts, provide significant support for modular development. 'Modularity-in-the-small' is achieved through building programs using small single-purpose functions, and where possible taking advantage of the non-strict semantics of a language. In this section we describe the various language-specific mechanisms for supporting 'modularity-in-the-large'.

SML provides significant support for modular programming. It has separate constructs for module implementation, *structures*, and module interface, *signatures*, which enables reuse either through attaching various signatures to a single structure or associating a single signature with multiple structures. Each SML structure provides a default signature, everything in the structure, which is overridden by any explicit signature association. SML's signatures provide detailed syntactic information for potential users of an associated module, and type abstraction can be achieved through assigning an *opaque* signature to a structure. SML's modules are not first class but are supported by an extension of the core language. However they can be used to create new modules either simply through containment or through the application of *functors* to existing modules. These parameterised modules are also part of the extended language.

Haskell's modules are largely used as a name space control mechanism. Implementation and interface details are provided by the same entity, whose export list names those entities that are available to any potential client. This list is devoid of any type signatures. A module's interface can also be declared when the module is used, but is constrained by the interface declared by the module. Haskell's module system also provides a means of creating abstract types by specifying a type constructor without its value constructors in a module export list.

We illustrate the Haskell module system with two simple modules `Exp` and `Imp`. `Exp` includes a declaration of a type class `ExpTC`, a data type `ExpT`, and an instantiation of the type class. All of these entities are in the interface of the module including the value constructors `Con1` and `Con2` of the data type. Thus the data type `ExpT` is not abstract when used by any client of the module `Exp`. Module `Imp` imports the type class `ExpTC` from the module `Exp` and declares a data type `ImpT` which instantiates the imported type class. The type is abstract to any client since it is presented in the export list without its value constructors.

```
module Exp (ExpTC, ExpT(Con1, Con2), expFun) where
         data ExpT = Con1 Int | Con2 Bool deriving Show
         class ExpTC a where
                 expFun :: a -> a
         instance ExpTC ExpT where
                 expFun = id

module Imp (ImpT, expFun, createImpT) where
        import Exp(ExpTC, expFun)
        data ImpT = Con (Int,Bool) deriving Show
        createImpT = Con (0,True)
        instance ExpTC ImpT where
                expFun = id
```

Nicklisch and Peyton Jones [90] describe how SML's substantive support for modularity can be largely expressed in Haskell using its module system.

Clean also provides a robust environment for modular programming which is similar to that of Modula-2 [146], where module implementation and module interface are provided by distinct constructs, an *implementation module* and a *definition module*, but each implementation has at most one interface. Clean's module-based abstraction support is similar to that of Haskell.

Although Miranda does not have an explicit module construct, modules are delivered through Miranda scripts (files). That is, a Miranda script can be viewed as a module. The Miranda `%export` directive provides interface control which is used when a script is imported into a client script. That is, modular development in Miranda is supported by defining program entities in different scripts, and enabling reuse through the language's file import/export mechanism. Miranda also supports parameterised scripts where definitions rely on information provided when the script is used by a client script.

FAD's modelling language includes the macro units *subsystem* and *module*. These units support a hierarchical approach to managing the development of a large system. A system can be divided into several subsystems which are developed independently but to known requirements. A subsystem is further divided into several modules each of which should be a cohesive unit with a clear, specific purpose. External access to a subsystem's or module's entities is mediated through an *exclusive signature* associated with the host macro unit. Descriptions of FAD's macro units, exclusive signatures, and the various relationships between units are presented in Chapter 5.

In conclusion, object-oriented software development as described in Chapter 2 is guided by modularity. That is, modularity drives functionality. The reverse is true in the functional paradigm and therefore when developing using FAD. One first describes the functional requirements of a system and then builds a modular system which supports them in as effective and efficient manner as possible. The main reason for this is that in an OO system, objects (or modules) are first-class and are therefore the fundamental building block upon which a system is developed. Functional programming has first-class functions, and modules are used to aid the management of development.

In the final section we describe the recent influence that *monads* have had on software development within the functional programming paradigm.

3.2.4 Monads

Monads are a recent addition to the functional programmers' toolbox. They encourage a structured and sequential approach to program development, and have resulted in a new approach to interactive programming in pure, non-strict, functional languages [139, 138, 140, 103].

Although monads have their roots in category theory where they are sometimes referred to as triples, one does not have to be a category theorist either to understand their structure or to practise their use. For the purposes of functional programming, the simplest view of a monad is as a unary type constructor (commonly called `m`) accompanied by a pair of polymorphic functions. One function (variously called `unit`, `unitM`, `return`, or `result`) takes a value of a particular type, and creates an item of the monadic type. The other function (variously called `bind`, `bindM`, `then`, `(>>=)`, or `(*)`) takes an item of the monadic type and a function from a value (wrapped in the first monadic type) to another monadic type, and returns an item of the second monadic type. From now on we will view a monad as the triple `(m, return, (>>=))`, where `return` and `(>>=)` have the following type specifications:

```
return :: a   -> m a
(>>=)  :: m a -> (a -> m b) -> m b
```

Haskell 98 provides a monad constructor class that includes additional function specifications to those presented above.

Another description of a monad is as a type of computations so that `m a` is the type of computations (of a certain sort) of values of type `a`. With this view in mind, `return` turns a value into the computation that simply returns the value. `(>>=)` takes a computation which returns a value of type `a`, extracts the value returned by the computation, and applies the second (functional) argument to this value which returns a computation that returns a value of type `b`. In essence programming with monads replaces functions from *values to values* by functions from *values to computations*, where the notion of a computation has several different interpretations such as one that does some I/O.

Monadic I/O is part of the Haskell language definition, and compares favourably to the other functional I/O alternatives, *dialogues* or *continuations* [53]. Peyton Jones and Wadler describe how the type `IO a` integrates the functional world with the non-functional world (pure and impure) [103]. The functional world is all about *being*, in that an expression in a functional language *denotes* a value. In contrast, the imperative world in which IO more naturally sits, is about *doing*, and an IO command should *perform* an action. Thus the type `IO a` in the words of Peyton Jones and Wadler denotes actions that, *when performed*, may do some I/O and then return some value of type `a`.

One of the main design implications of monad use is that it encourages encapsulation and programming through an (monad) interface. Encapsulation prevents any changes to code from having a rippling effect through the software, and simple interfaces make explicit how one can combine program components. Monadic development enforces a particular evaluation strategy that is sequential in nature. For any function defined over a 'monadic' datatype - a datatype with associated monad combinators - the computation will be sequential and guided by the combinators `(>>=)` and `return`. That is, one abstracts over the computation as opposed to the more common approach of abstracting over the particular data representation.

We present two illustrative examples. The first is the function `exIO` which illustrates monadic I/O that looks very similar to the code one would write in an imperative language like C [72]. The second example presents the function `allSquarePlusOne` which takes a list of integers, each of which is squared and then incremented.

```
exIO :: IO ()
exIO = getChar          >>= \ c1 ->
       getChar          >>= \ c2 ->
       putChar c2       >>= \ _
       putChar c1

allSquarePlusOne :: [Int] -> [Int]
allSquarePlusOne xs
        = xs                    >>= \ x ->
          return (square x)     >>= \ y ->
          return (y+1)
```

`getChar` and `putChar` mimic the C functions `getC` and `putC`, and `exIO` clearly illustrates the sequential nature of functions defined using monads. That is, `exIO`:

takes a character from the standard input and binds it to c1. It then takes another character from the standard input and binds it to c2. c2 is then sent to the standard output and the non-existent result bound to a wildcard. Finally c1 is sent to the standard output.

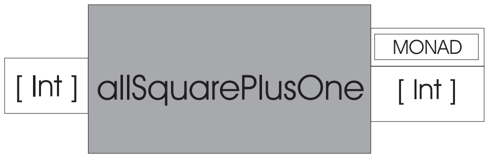

Figure 6: Monadic Function

The design of `exIO` is very similar to the design one would use in an imperative language like C. This is an excellent illustration of one of the main benefits of monads: the ability to mimic impure features without losing all the benefits of pure, non-strict, functional programming such as referential transparency, higher-order functions and lazy evaluation. In addition, quite disparate functions can be described using the same computational abstraction.

As with other forms of encapsulation, modification of code can be achieved relatively painlessly, and more importantly, locally. A large scale example of monadic software design is the Glasgow Haskell compiler, itself written in Haskell. The compiler uses monads for various bookkeeping tasks, and when the type checker needed to be updated to maintain information about the current line number, this was not an onerous task [104].

FAD supports monadic development through a permissive signature `MONAD` which specifies the monadic combinators. Thus `allSquarePlusOne` is represented in FAD as in Figure 6.

3.3 Summary

In this chapter we have described an approach to software development which is significantly different to that practised in other paradigms. We described the following constructs which have a significant impact upon software development in the functional programming paradigm:

- *functions* are the fundamental building blocks of the paradigm. Their output depends solely on their input;

- *types* are sets of values that provide guidance through all the stages of software development. They make explicit the values that are acceptable as arguments for a function, and those that will be returned by a function;
- *type constructors* construct the aforementioned types. They may take one or more arguments to construct a type. Type constructors can be categorised by their *kind* which specifies the number and form of parameters required by the type constructor;
- *value constructors* construct values of a type. They also may take one or more arguments;
- every functional programming language provides support for the development of *user defined types* through the algebraic type mechanism. A new type is created through the declaration of a new *type constructor* and its associated (new) *value constructors*;
- *abstract types* are types whose construction details - value constructors - are invisible to potential clients;
- *parametric polymorphism* is the predominant form of polymorphism supported by the paradigm. It enables a function to be reused over several types;
- *permissive signatures* are FAD units (fully described in Section 5.2.3) that specify entities that are defined over an associated type(s). They provide a behavioural guarantee for an associated type and can be implemented as type or constructor classes in certain languages. A permissive signature is not an interface to a type but rather states the *minimum* functionality defined over the type; and,
- *exclusive signatures*, which are also FAD units (fully described in Section 5.3.3) that specify an interface to an associated macro unit such as a module. Where a permissive signature states *at least this*, an exclusive signature state *only this.* They play an important rôle in developing software based on *abstract types.*

Functional programming is different to other paradigms in that:

- mutable variables are replaced by values;

- procedures and methods are replaced by functions whose output depends only on their input; and,
- encapsulation for data protection is replaced by encapsulation for modularity.

Thus software is developed through functions that delegate their behaviour to simple functions with a clear singular purpose. That is, one is encouraged to develop functions using simpler functions that implement a required behaviour. How this behaviour is implemented is not of interest to the client function. Large systems can be built using the support for modularity-in-the-large. Abstract data types provide a mechanism for modular design based on information hiding.

A different approach to software development requires a different approach to modelling systems, which in turn requires new modelling languages and methodologies. In the following chapter we describe methodologies and their languages, emphasising the benefits of their application. In Chapter 5 we describe the modelling language of FAD, and in Chapter 7 the methodology and its techniques.

Chapter 4

Analysis and Design Methodologies

The previous two chapters have described the object-oriented and functional programming paradigms with an emphasis on their different approaches to software development. Chapter 2 also included a brief overview of object-oriented analysis and design methodologies and how in combination with object-oriented languages they can deliver a packaged approach to software development. In this chapter we present a description of analysis and design methodologies (ADMs) as a modelling language and set of integrated techniques that deliver models using elements of the modelling language. In Section 4.2 we outline the benefits of using an ADM within a software development project. These include using a language whose purpose is modelling problems and solutions rather than implementing them. In Section 4.3 we will describe the benefits of adopting a packaged approach where the ADM and implementation language are from the same paradigm.

4.1 Analysis and Design Methodologies

Sutcliffe [134] argues the case in favour of ADMs as follows:

- Before building systems we have to understand them.

- To understand systems we should make a model.

An analysis and design methodology is a medium for understanding a problem, modelling a solution, and managing and documenting software development. Each methodology is a combination of a modelling language, typically with associated graphical notation, and a collection of integrated techniques which support analysis and design.

The development of large software systems requires three forms of management. Firstly, there are a collection of *project management* tasks which include the general management of multiple development teams, ensuring that deadlines are met within budget, and that resources are available and accessible. Secondly, *documentation management* is integral to successful software development. System entities and decisions should be documented and made available for current and future reference. Finally there is *development management* which may involve the application of an in-house or named development method. Within this thesis and with FAD, we will focus solely on the final two forms of management since project management can be delivered independently of any particular methodology.

In the following two sections we present the essential elements of a methodology - its modelling language and techniques.

4.1.1 Modelling Language

Every ADM has an associated modelling language through which systems are modelled and documented. In most cases the modelling language will support both graphical representations and textual descriptions of its units and their interactions. Thus, structured methodologies such as SSADM [41] and SA/SD [37, 153, 152] have modelling languages which deliver, for example, *data flow diagrams* and *logical data structures*. Every language unit, such as *process*, and *data store*, will have a clear definition and associated graphical representation. That is, most modelling languages, in common with implementation languages, have a defined syntax and semantics. Typically the semantics of a modelling language are described informally.

The modelling languages associated with OOADMs include elements which represent the OO building blocks - *classes*, *objects* - and their various associations. Each description of a class includes class responsibilities and details of associations with other classes. Class documentation has similarities to CRC (Class,Responsibility,Collaboration) cards [8] where one presents the class name, followed by a list of responsibilities and then any links to other classes in the system.

A modelling language is specifically used for modelling systems and not for implementing systems. Although one can (partially) generate source code using CASE tools such as Rational's Rose [33, 34], a modelling language should aid development of an implementable solution, and not provide full details of a specific implementation. Therefore a graphical modelling language is not a visual programming language such as Prograph [110], or Visual Haskell as put forward by Reekie in his thesis *Realtime Signal Processing: Dataflow, Visual, and Functional Programming* [112]. Modelling languages are typically smaller and semantically less rich than their implementation language counterparts since abstractions take precedence over detail. There are however benefits in having a correspondence between the modelling language and potential implementation language. This correspondence is maximised when the modelling language and implementation language are of the same paradigm.

A modelling language is in essence an abstraction of an implementation language, where one focuses on the essential features of the paradigm whilst disregarding the elements that are only required by an implementation language. Most graphical modelling languages do however support the embedding of either implementation language code or pseudocode into their models. For example, one typically records a method within a class using the syntax of an OO language. The techniques of a methodology take as input and return as deliverables models developed using the modelling language.

4.1.2 Techniques

Each methodology provides the user with a collection of integrated techniques. With action-oriented, structured development the techniques focus on delivering data flow-centric descriptions of the system, which are refined top-down into more detailed descriptions. The models are typically presented as data flow diagrams, logical data structures and structured English, or graphical representations of process dependencies built via the three common imperative constructs: *sequencing*, *selection* and *iteration*. In data-driven approaches such as Jackson System Development [62], the modelling components are similar but the techniques guide the developer in building processes which reflect the structure of the system's data, such as files.

OO methodologies encompass techniques that describe the problem in terms of abstractions which encapsulate their state. The various analytical models will be iteratively modified through a collection of techniques that return an implementable solution. The models delivered include:

- models of the major classes and their various associations;
- the objects that collaborate to deliver some specified functionality. These models include the messages passed between objects; and,
- models of classes with significant state dynamics.

Most modern methodologies support both analysis and design. Analysis techniques focus on developing models of *what is required*, where design techniques deliver *how it is achieved*. That is, analytical techniques tend to be problem-centric, reflecting what is required without imposing any design choices. The results of the analysis phase are delivered to the design phase, where techniques manipulate the models to deliver an implementable, maintainable and potentially reusable design.

Each technique will have a clear purpose, explicit input requirements and a set of deliverables. For example, *use case analysis* which is an essential analytical tool of use in OOSE [64], the Booch Method [15], and supported by UML [16], is a methodical approach for gleaning information from the requirements of a system. It produces a collection of scenarios that can be used in the development of classes and their collaborations. CRCs can then be used as a technique for analyzing the scenarios returned by use case analysis. Similarly the *entity action step* of Jackson System Development [62] aims to produce an abstract description of the real world using only interdependent nouns and verbs as the medium. The *entity structure step* takes such a description and delivers models of the life span of each entity.

Methodologies which encourage a strictly linear application of their techniques typically have models that are linked to a particular phase of development. For example, SSADM's *effect correspondence diagrams*, which identify effects caused by a single event, are produced midway through the process. They are developed from existing *logical data structures*, which present a static view of the system's data and interrelationships. Methodologies that encourage iterative and incremental development tend to have a set of models (and associated diagrams) that are of use throughout system development.

Whether practised iteratively or linearly there are several benefits in using an ADM to support software development, which are described in the following section.

4.2 What are the Benefits of Using an ADM?

Birrell and Ould [12] present the following argument in favour of using a methodology during software development.

> Anyone undertaking software development, on no matter what scale, must be strongly advised to establish a methodology for that development - one or more techniques that, by integration and control, will bring order and direction to the production process.

We will present the reasons for using an analysis and design methodology as an aid to successful software development in the following sections. The first describes the benefits of using a language whose raison d'être is modelling rather than implementing an efficient solution.

4.2.1 A Language for Modelling

Each methodology delivers a collection of models using the units and relationships defined in its modelling language. Since a *model* is an abstract representation of a design or specification, a modelling language is a collection of elements that support the construction of an abstract description of a problem or solution. Thus one can produce models of a system or design that emphasize the major abstractions involved whilst avoiding the unnecessary details required in implementation language code.

A modelling language enables development which reflects best practice in a paradigm rather than best practice due to the idiosyncrasies of a particular implementation language. A modelling language thus enables a separation of concerns, by allowing the designer the freedom to develop systems beyond the constraints enforced by the nuances and eccentricities of a particular programming language. That is, the implementation language does not drive design but instead enables a design to reach fruition. For example, an FP design may require a type that provides an explicit interface to potential clients. Abstract data types provide an explicit interface but their implementation is not uniform across the languages of the paradigm. For example:

- in Haskell one declares a type in a module which does not export any of the type's construction details;
- in SML one can either use the `abstype` mechanism or declare the type in a structure which is associated with an opaque signature; and,
- in Clean one specifies a type constructor without its value constructors in the definition module associated with the implementation module within which the type was declared.

It is not relevant to a design whether one declares an ADT through a module interface, or whether the implementation language has a keyword to indicate such a construct. For design purposes one simply requires a clear model of an abstract type and information regarding what one can do to values of the type.

A methodology's modelling language provides accessibility to a system's design to those who have an interest in the system but are not familiar with the (potential) implementation language(s). A graphical representation of a design typically presents a clearer picture than several pages of code, and most modelling languages support several orthogonal views of the same system.

In the following section we describe how a methodology delivers an integrated set of techniques that deliver models using the elements of the modelling language.

4.2.2 Development Guidance Provided by a Set of Techniques

An ADM can aid the development of large systems by providing a collection of integrated techniques that guide and drive the development process. There are parallels here with the use of operational research (OR) techniques to aid business decisions. OR techniques encourage the user to look at a problem at a level of abstraction that would not otherwise be achieved. They also offer a set of well-defined steps that enable the user to break the problem down into understandable pieces, and then to put them back together again in the most effective way. ADMs mimic this process. They cannot guarantee the best design, but they can improve one's chances of achieving an effective and acceptable design.

One of course must be careful not to make any false claims. There is no statistical evidence that a particular methodology outperforms others, or that methodology use has significantly improved performance. Such research is difficult to perform for many reasons including problem consistency, costs of failure and so on. However, modelling processes are used in other fields with evident success and there is no obvious reason to dispute their transferability to software development.

Each ADM provides a *template* for development built on a collection of techniques. How strictly one adheres to the template will depend on the type of problem and one's familiarity with the problem domain. The collection typically includes:

- techniques for discovering the essential data and functionality requirements in the problem and for representing them using elements of the modelling language;
- techniques for analysing the data and functional requirements and modelling them in terms of collaborating elements;
- techniques for dividing the system into manageable units (components) which can be developed independently;
- techniques for describing the system in terms of its major components and their interactions; and,
- techniques for translating models of the problem into models of the solution.

Thus beyond the support for discovering the required data and procedures, methods or functions, most modern ADM's also include techniques for developing large systems through giving guidance on how to divide a system into sensible components. This division is normally directed through one or more criterion for modular development. The modelling language of each ADM will provide elements that present the modular architecture of a system, one of several system insights or views that can be described.

4.2.3 System Viewer and Complexity Manager

All ADMs support several views of a system both during development and upon completion. Where an implementation language presents one view of the system based on the syntax of the language, ADMs provide some or all of the following:

- a *static* view which represents the major data elements of the system and their relationships;

- a *functional* view which describes system functionality;
- a *dynamic* view which focuses on the effects of events on a system entity; and,
- a *modular* view which describes the high-level architecture of the system.

Thus an ADM is a medium for communicating a design in various formats typically using graphical notation. In the object-oriented paradigm, design patterns [27, 113, 49], are becoming an increasingly popular means of sharing effective and reusable designs. A pattern is

> the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts. [113]

These abstract patterns, could not, and more importantly, should not be presented in any of the many object-oriented implementation languages. Through graphical representations using modelling language notation (Gamma et al. use OMT, Objectory and the Booch Method notation [49]), accompanied by some commentary on their development, one can present a clear picture of a pattern that can be understood by any OO practitioner.

Beyond presenting clear views of a system in development, a methodology will encourage the user to produce a thorough collection of system documentation.

4.2.4 System Documentation

Although most modern implementation languages provide mechanisms for accompanying code with some commentary, this tends to only support a description of the terminal construction of the software. There is no obvious site for a historical description of development or non-implementation documentation.

An ADM will support full documentation of the entities of the system, which can include a development history for a particular entity or a snapshot of a system or component of the system during development.

In conclusion, most ADMs provide generic support for system development which can be categorised as in the previous sections. However, modelling a system using any methodology and implementing its design in any language is not advised. Instead one should look to analyse a problem, and design and implement a solution using tools of the same paradigm. We present the reasons for this recommendation in the following section.

4.3 Paradigm-Consistent Approach to Development

Most analysis and design methodologies can be classified by the paradigm they support. The Booch Method [15], OMT [120], Coad-Yourdon's OOA/OOD [28, 29] can all be classified as object-oriented methodologies. UML [46, 16] though not a methodology, is a notation for supporting object-oriented analysis and design. Similarly, SSADM [41] and SA/SD [152, 153] are structured development approaches which naturally support the constructs found within the imperative/structured paradigm.

The paradigm associations of each methodology are not coincidental. Structured methodologies were introduced in response to perceived faults in the systems developed using imperative languages. They encourage a particular approach to system development and construction through their concentration on data flows, and the stepwise refinement of system processes which are developed using the procedural programming constructs, iteration, sequencing and selection. That is, they are fundamentally underpinned by imperative constructs. Object-oriented methodologies, in common with object-oriented languages, naturally support object-oriented development. Although each methodology has its own notation and specific set of techniques, they each support the development of object-oriented systems.

Coad and Yourdon [29] argue that

> It was difficult to think about structured programming when the languages of choice were assembler and FORTRAN; things became easier with Pascal, PL/1, and ALGOL. Similarly, it was difficult to think about coding in an object-oriented fashion when the language of choice was COBOL or plain-vanilla C; it has become easier with C++ and Smalltalk.

Of course one can always implement a 'paradigm A' design in a 'paradigm B' implementation language but not without development costs. Rumbaugh [120] claims that object-oriented designs can be implemented in non-object-oriented languages but the programmer will be required to: translate classes into data structures, pass arguments to methods, allocate storage for objects, implement inheritance in data structures and so on. Booch [15] is more dismissive, arguing that

> object-oriented analysis and design is fundamentally different than traditional structured design approaches: it requires a different way of thinking about decomposition, and it produces software architectures that are largely outside the realm of the structured design culture. These differences arise from the fact that structured design methods build upon structured programming, whereas object-oriented design builds upon object-oriented programming.

A methodology that builds upon functional programming also requires a different design approach, and should be built from the underlying abstractions of functional programming. Brooks [20] describes a mismatch of paradigms as an example of *accidental complexity* which adds to the underlying *essential complexity* of software development. The degree of essential complexity is a function of the type of problem and familiarity with the problem domain, and thus cannot be avoided. Accidental complexity can be avoided by adopting a paradigm-consistent approach from the modelling of requirements through to the design and implementation of a solution.

Simply using a combination of stepwise refinement, data flow design and a general modular approach will ignore the specific benefits of programming in a functional style with a functional language. Equally so if one models the problem as a collection of interacting objects that communicate with one another through their interfaces.

4.4 Summary

This chapter has outlined the structure of ADMs and the benefits of their application within a software development project. Although modelling in itself is a constructive practice, modelling using elements that are familiar to a potential implementation language enhances the applicability of its products. We therefore believe that there are strong arguments in favour of a functional ADM that supports the essential features of the functional programming paradigm, and whose language units are accompanied by a graphical representation. In the following chapters we describe the modelling language and techniques of FAD, an analysis and design methodology that supports software development within the functional programming paradigm.

Chapter 5

FAD Modelling Language

In the previous chapter we outlined the benefits of using analysis and design methodologies (ADMs) as aids to software development. In addition we argued that the best results are achieved when the ADM and implementation language support development within the same paradigm. That is, one can argue at length regarding which paradigm provides the best support for software development, but one achieves the most natural, efficient and effective development package when one remains within a single paradigm from problem description through to implementation and delivery.

Chapters 2 and 3 described and contrasted the object-oriented and functional programming paradigms. Although they have their similarities there are clearly significant differences. These differences impact on the software designs of each paradigm and a cost is incurred if one attempts to switch paradigms between any phases of development. The object-oriented and structured paradigms have several ADMs which support a complete development package within their paradigm. We believe that the functional programming paradigm requires methodologies to support its software development approach.

In Chapter 3 the major building blocks and glue of the functional programming paradigm were described. In this chapter we describe the modelling language of FAD (Functional Analysis and Design). We believe that any paradigm-specific ADM should support, in a natural manner, software development within the paradigm with minimal notational overhead. In addition, a paradigm-specific ADM should not reinvent or over-constrain the software development process but should reflect and encourage common practice. This requires a modelling language that supports the major building blocks and glue of the paradigm with a minimal collection of graphical notations for pictorially representing analytical and design models. The methodology should support the recording and storing of entities in a manner that eases use and maximises discovery of potentially reusable entities.

FAD is both a modelling language and a set of techniques to support software development within the functional programming paradigm. FAD should be practised within an iterative and incremental development process. This is facilitated by adopting a single set of notations and diagrams that are applicable throughout development. That is, one does not use particular types of diagram and entity representations at particular stages of development and then convert them to new diagram types and representations applicable to later stages as is the case with most structured methodologies such as SSADM. Any FAD diagram and its constituent notation is of use throughout the development process but will be iteratively updated in step with iterations in the system design. FAD diagrams include:

- function dependency diagrams which present a function with those it uses in its implementation;
- type dependency diagram which provides the same service for types; and,
- module dependency diagrams which present views of the module architecture of the system.

FAD supports development in any functional language and not in a specific language. It therefore needs to support constructs that are common to all functional languages, or shared by just a few. In Section 5.2 we describe the basic units of the language. We divide them into the micro units: types, functions and permissive signatures, and the macro units: exclusive signatures, modules, subsystems, projects, and files. We provide both informal definitions of the units and their FAD notation. For each unit we provide a brief qualification for the chosen notation. Each type of unit has an associated *Unit Description Document* in which one can record the unit's name, version and other relevant information. These description documents provide an historical record of the development of a particular component of a system.

In Section 5.4 the inter-unit relationships supported by FAD are described. These include: the type use relationship, function use relationship and associations between types and permissive signatures, modules and exclusive signatures, and subsystems and exclusive signatures. Once again the informal definition is accompanied by a description of the FAD notation which includes some commentary on the choice of notation. In describing the units and relationships of the modelling language we present the diagrams of FAD that afford various views of a system.

In Chapter 6, we demonstrate how common functional constructs are defined and represented in FAD. In the following section we present a case study that will be used to illustrate elements of FAD's modelling language and the application of its techniques.

5.1 Case Study

The case study was chosen because it is both small enough to comprehend fully and large enough to illustrate the various components of FAD. A larger case study - a CASE tool consistency checker - is presented in the appendix to this thesis. A system is required to automate the production of various football league related data. The system stores current data on the league's football teams, the teams' players, historical data on league tables, results, and scoring tables. New results are entered by a data entry clerk and, upon request, a current version of the league table or scoring table is generated.

In brief, the system must support the following functional requirements:

- the inputting of football results (for as many leagues as required);
- the production of league tables;
- the production of scoring tables which present the top scorers in the league, their team, and the number of goals scored;
- the production of attendance tables which present teams in order of average home attendances;
- the transfer of players between teams;
- the updating of team data due to recent results; and,

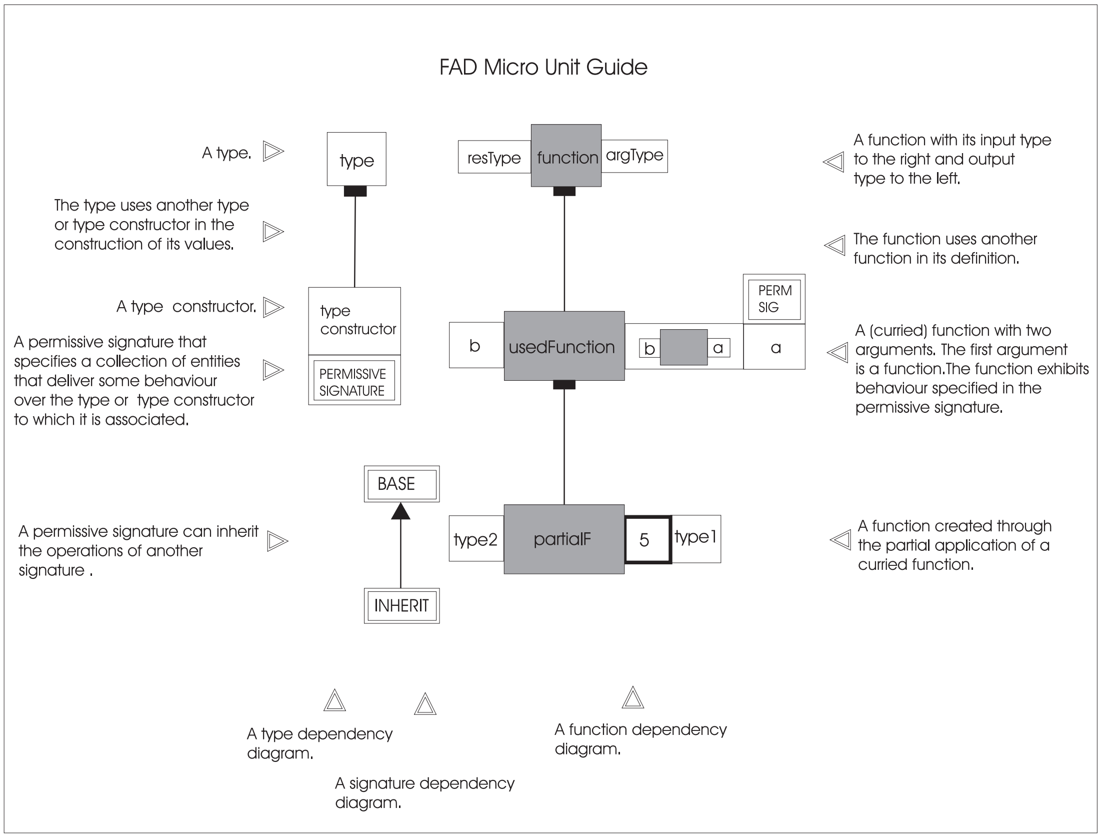

Figure 7: Micro Unit Guide

- the updating of player data due to recent results where the data include appearances and goals scored.

No non-functional requirements are stated and the system should be developed so that if new functionality is required, it can be introduced at a minimum cost.

5.2 FAD Micro Units

The basic micro units of FAD are types, functions and permissive signatures. We describe each in turn and then describe how they can be combined to support common constructs of the functional paradigm. Figure 7 presents a 'Micro Unit Guide' which summarizes the micro units, their relationships and diagrams.

The diagrams built using these units and relationships have a (informally) declared syntax and semantics. These are described in the following sections and in Sections 5.3

and Section 5.4 where we present the macro units and relationships of FAD. Each unit and relationship is illustrated by an example from the case study.

5.2.1 Types

A type is a collection of related values which have some common usage. Examples include the type of characters and the type of Boolean values. A type typically has a mnemonic name that reflects the characteristics of its values. Modern functional languages support type aliases which assign a name to a type which is appropriate in a given context. In FAD each type has a unique name which begins with a lower case letter. In Section 3.1.2 we emphasised the importance of types to software development within the functional programming paradigm. They provide a specification of a program's entities, and enable the early detection of errors.

Every functional language (and other typed languages) provides a set of built-in basic types whose values are primitive to the language. Most languages provide characters, Booleans and various collections of numerical values as basic types. They are typically accompanied by built-in functions and operators defined over the types. These normally include the arithmetic, relational and logical operators.

The languages also support built-in and user-defined composite types whose values are constructed using values of existing types. Tuples and lists are usually provided by a functional programming language. In common with basic types the languages provide functions and operators defined over these built-in composite types, such as list construction operators, and pair selection functions.

Types can be constructed by users through the multi-purpose *algebraic type* mechanism using a unique *type constructor*. The values of an algebraic type are constructed by using one of the *value constructors* declared with the type. Enumerated types, sum types, product types, parameterised types and recursive types can all be declared using the same mechanism as described in Section 3.1.5.

Abstract data types, which provide a mechanism for modular development through information hiding, are supported by all modern functional languages. Recently there has been much interest in existential types [74] as a mechanism for implementing first-class abstract data types. FAD's support for these and for tuple types, record types, algebraic types and abstract data types are left to Sections 6.10, 6.1, 6.2, 6.3 and 6.4

Type Description Document	Football
Constructor Name:	`teams`
Version:	19990620:0
Kind:	`*`
Module:	
Types Used:	`date`, `team`, `collection`
Parameters:	
Permissive sigs.:	`TEAMSCON`
Description:	

The type of football teams. Each team must be accessible and their information updateable. The date represents the latest update to the teams' data. Each team will include data on its results, attendances and other team-related information.

Figure 8: Type Description Document for the Type `teams`

respectively.

The details of a type are described in a *Type Description Document* (TDD) as illustrated by the TDD for the type `teams` presented in Figure 8. Each type may have several TDDs illustrating the iterative development of the type. However each type will have a TDD which represents the current form of the type which will be the chronologically most recent version determined by the version number.

Each type description document presents a description of a type. To the right of the header is the project within which the entity is defined. The list in the body of the document presents the following information:

- the name of the type constructor of the type which begins with a lower case letter;
- the version of the type denoted by a `date:natural number` value to accommodate multiple versions in a single day;
- the kind of the type constructor. Type constructors with the kind `*` are simply

types;

- the module in which the type is declared. Every micro unit is declared in a unique and identified module. The organization of modules and their entities is a design decision and therefore the containing module will typically be recorded in a later version of a TDD;
- the types used in constructing values of the type;
- the type variables and associated permissive signatures. This entry will be blank for any type whose type constructor is of kind `*`. We write `PERMSIG a` for each type/permissive signature contract association. This association is described in Section 5.4.4. A type variable is written as a single lower case letter. This name has no intrinsic value and if there are no associated permissive signatures then no entry will be recorded;
- the permissive signatures instantiated by the type (see Section 5.4.4). The parameter (or a parameter for permissive signatures with more than one parameter) of the permissive signature must have the same kind as the type constructor of the type. Each algebraic type instantiates a *constructor signature* as described in Section 6.3;
- a textual description of the type.

In summary, a type description document is a host for information relating to the development of a type. As a type is iteratively developed the document will be updated to reflect design decisions. The document is stored in FAD's data dictionary as described in Chapter 8.

FAD Notation

Types and values of types are represented in FAD by similar notation. A type is represented in FAD by a rectangle (or box) enclosing the type's name as illustrated in Figure 9. This notation was chosen because a type is a collection or box of values with some common characteristics. Alternatively, one can view types as a mechanism for partitioning the universe of values (ignoring some overloading of numeric literals), and partitions are often represented as rectangular segments of a set.

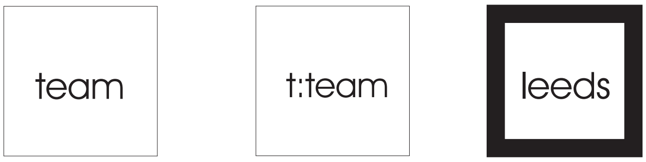

Figure 9: A Type, Parameter of a Type, and a Named Value of a Type

If the type is constructed through the application of a non-nullary type constructor to one or more types or is a parameterised type, the name of the type must include the type constructor and the name of the parameters. Typically the constructor name will prefix the parameter names. A type variable is simply represented by a lower case letter.

One can also add a name to a particular use of a type to make explicit how a type's value is being used in a particular context. That is, parameter names or type values in the form of a valid expression or literal, or names associated with a value can also be included in a type rectangle. A parameter is written `parameterName:typeName` or simply `parameterName` if the type is clear due to the context, and a value can similarly prefix a type name or appear on its own. The rectangle enclosing a value of a type has a thick solid perimeter. This notation differentiates a type from a value of a type but with minimal added notational overload. One can use this value notation to represent *partial application*. This is described along with FAD's support for the curried form of multiple argument functions in Section 5.4.3. Hence one is able to reuse the same notation for a type, a type variable, a non-nullary type constructor, a named parameter of a type and a value of a type or a name associated with a value. A functional type has its own notation as described in the following section.

5.2.2 Functions

The major building blocks of the functional paradigm are pure functions that map values from a single type (argument type) or multiple argument types, to a value of another type (result type). Functions are first-class citizens in functional languages and therefore can be arguments of other functions, be returned by functions and be components in data structures. Functions can be created statically or dynamically through the application of a curried function to an incomplete set of arguments known as *partial application*

FAD supports all forms of function use including functions that are defined using subsidiary functions (see Section 5.4.9), functional arguments (see Section 6.9) and functions with multiple arguments in the form of *curried functions* (see Section 5.4.3).

The details of each function are described in a *Function Description Document* (FDD) which in common with TDDs will be developed iteratively. We present in Figure 10 the FDD for `getData`, the higher-order polymorphic function which takes a functional argument and a value of the type `teams`, which is a collection of values of type `team`, and returns the result of applying the function to each `team` value. The return type is `collection a` where the type `collection a` is used by the type `teams`.

In common with TDDs, the collection of a function's FDDs describe the iterative development of functions. That is, the collection of documents presents a record of design decisions for a particular function. These are of potential use in future maintenance of the system, in supporting reusable designs and to allow rollback within an iterative design framework.

The project within which the function is defined is presented to the right of the document's header. The list in the body of the document presents:

- the function's name which begins with a lower case letter. An operator name is presented in prefix form enclosed in parenthesis. Function names are not necessarily unique since mnemonic identifiers are encouraged in order to support the discovery of abstractions such as polymorphic functions and overloading. However no two functions with the same type specification will have the same name. This also prohibits the co-existence of a polymorphic function and its monomorphic instantiations. In Chapter 7 we describe a technique for developing polymorphic functions that includes the removal of its monomorphic counterparts. Functions with the same name can be discriminated by qualifying their name with the name of the module in which they are defined. For example, the function `getData` can be qualified as `TeamsMod.getData`. This naming convention can be applied to any module entities;

- the version of the FDD represented by a `date:natural number` value;

- the module within which the function is declared (see Section 5.3.1). During the early stages of development one may record a subsystem as host. This applies to micro units of any kind;

- the function's arity. This will be used as a key for storing the function in the data dictionary as described in Chapter 8;

- the function's type specification written using the function type operator `->` ;

- the required type/permissive signature associations. This information will help guide development of the function and its associated types. For polymorphic functions, the permissive signatures provide constraints on the types that can instantiate the associated type variable. Type/permissive signature contract associations are written `PERMSIG a` where `a` is the name assigned to the type variable. For multiple parameter signatures the signature name is followed by the requisite number of type variables. See Section 5.2.3 for a description of permissive signatures, and Section 5.4.4 for an explanation of the type/permissive signature association. This information provides another means of selecting functions for potential (re)use;

- the required type/signature instantiation associations. These are written as above replacing the variable name with the type name. For example, `PERMSIG typeName`. Many functions will initially be developed as monomorphic functions. Any associated permissive signatures will provide information regarding the behaviour of the function. They will also provide constraints on the implementation of any associated types, and suggest potential function overloading when implementing in a language with such support. A type or used type may be required to instantiate a particular signature. This information aids the discovery of potential polymorphic and overloaded functions as described in Section 7.3.2;

- the non-argument functions used in the definition of the function. Each function is presented with its type specification to distinguish overloaded function names. A colon separates a function name from its type. A function with conditional behaviour will not necessarily use all the functions. The function's *dependency diagram(s)* will clarify the dependencies as described in Section 7.2.2;

Function Description Document	Football
Name:	`getData`
Version:	19980810:1
Module:	`TeamsMod`
Arity:	2
Type Specification:	`(team -> a) -> teams -> collection a`
Contract Association:	
Instantiations:	`CONTAINER collection`
Functions Used:	
Description:	
This function retrieves data from a collection of teams through the application of a data-getting function to each team in the collection. The type teams is required to support 'mapping' behaviour.	

Figure 10: Function Description Document for the Function `getData`

- a description of the function.

The function description document provides significant information for developing a function and storing it in the data dictionary. The function's arity and permissive signature instantiations are used to store and retrieve functions for potential reuse. This approach is built on that described by Park and Ramjisingh [94] and An and Park [4], and is fully described in Chapter 8.

FAD Notation

A function is represented in FAD by a grey rectangle or box juxtaposed with its argument types to its right (consistent with function application syntax in all modern functional languages) and the result type to its left. The function rectangle is larger than the type rectangles. The grey box notation is motivated by the idea of a 'black

Figure 11: Function Representation

box' view of a function where one is only interested in the mapping between a function's inputs and outputs. Thus, one presents the type(s) of the input values and the type of the output value linked by a box whose inner details are not visible. The type boxes are external to the function box since it: conforms to the juxtaposition-based syntax between a function and its arguments found in most functional languages; it avoids potentially messy nested notation for the representation of permissive signature/type associations as described in Section 5.4.4; and, it simplifies the representation of functions with functional arguments or results.

A function name is written inside the shaded box, as is a functional parameter name if required. If a function has multiple arguments then its first argument appears next to the function rectangle, and each further argument appears to the right of each existing argument.

We illustrate FAD's function notation in Figure 11 where we present the FAD representation of the curried function `getData`. which takes two arguments. The first argument is of the function type `team -> a`. A function type is represented as a function with no name in the function box. When used as an argument or result type of a function it is enclosed in a type box.

5.2.3 Permissive Signatures

The development of functions and types requires as much information as possible. A function's development is guided by its type and required behaviour, and a type's development by the data and behaviour it needs to support. *Permissive signatures* provide a mechanism for specifying behavioural requirements.

Before describing permissive signatures we present an example which motivates their introduction and application. The function `getPlayer` takes a player's name and the collection of players of type `player`, and returns the relevant player. The function will test each player in the collection against the inputted name until a match is achieved.

If no match is reached an exceptional value is returned. The function therefore requires a test of equality of player names and needs to check each player in turn. These behavioural requirements can be made explicit through associating permissive signatures with the relevant type or type constructor. We associate the signature `EQ`, which delivers an equality testing function, with the type of players' names, `pName`. In addition, we associate the permissive signature `MAP` that specifies mapping behaviour, with the type `collection a` which is used by the type `players`.

A permissive signature declares operations that implement the behaviour indicated by the name of the signature. The operations are specified in terms of the parameter(s) of the signature. That is, a signature will have one or more parameters that are bound to the type constructors of the types that instantiate the signature. For example, the Haskell type classes `Eq` and `Ord` can be modelled in FAD as permissive signatures whose operations deliver equality and ordering behaviour. They are instantiated, for example, by the various numeric types.

Each entity may only be specified in a single permissive signature but can be reused in another signature through inheritance. Signature inheritance is described in Section 5.4.7. That is, since `(==)` is specified in the permissive signature `EQ` it cannot appear in any other permissive signature except through inheritance. Each permissive signature will be associated with one or more types that will *instantiate* the signature (see Section 5.4.4 for details on how this is achieved) such as the type `Int` and `Char` which instantiate the Haskell classes presented above. A type instantiates a permissive signature when bindings exist for each entity specified in the signature defined over the type.

A permissive signature provides a contract of usability for any type (or types when there is more than one parameter) which instantiate the signature. Each parameter will have an explicit *kind* where a kind identifies collections of type constructors in the same manner that types describe collections of values [66].

Each algebraic type instantiates at least one permissive signature which we call its *constructor signature*. In most cases the signature will have a single parameter that is bound to the instantiating type's constructor. Since most functional languages do not allow reuse of a type's value constructors, constructor signatures will generally be instantiated by a single type. That is, there will typically be a 1-1 correspondence between constructor signatures and algebraic types. The operations of a constructor

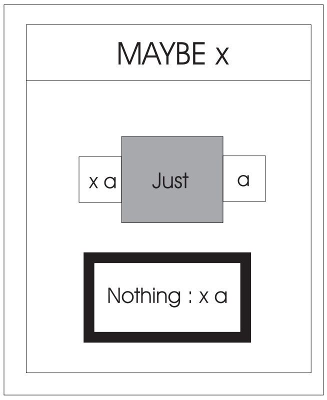

Figure 12: Constructor Signature

signature are the value constructors of an instantiating type. This is illustrated in Figure 12 where the constructor signature of the Haskell type `Maybe` is presented. The type has two constructors, `Just` which takes a value of any type `a` and returns a value of type `Maybe a` and `Nothing` which is a value of the type `Maybe a`.

The details of each signature are described in a *Permissive Signature Description Document* (PSDD) as illustrated by the PSDD for `EQ` which is equivalent to the Haskell class `Eq`. This PSDD is presented in Figure 13. In FAD, a name is associated with a type specification by writing the name followed by a colon and then the type.

The project within which the signature is defined is presented to the right of the header. The list in the body of the document presents:

- the signature's unique name which is written in upper-case letters;
- the version of the signature;
- the module in which the signature is declared;
- the signature's parameters and their kind. Although the parameter name is not important it must not clash with any type variable names that are not matched with the parameter. For example, a signature whose parameter is of kind `* -> *`, will possible specify operations over at least two type variables. One of kind `* -> *` will use the parameter name and the other of kind `*` must have a different name;

Permissive Signature Description Document	Football
Name:	`EQ`
Version:	19990317:1
Module:	
Parameter(s):	a : *
Operations:	(==): `a -> a -> bool`
(with type specs.)	(/=): `a -> a -> bool`
Inherited Signature(s):	
Description:	
This signature specifies the ability to test for equality over an instantiating type.	

Figure 13: Permissive Signature Description Document for `EQ`

- the signature's operations and type specifications. The specifications are written using the name of the parameter(s);
- the signature(s) from which the signature has inherited operations;
- a description of the signature.

Permissive signatures not only present the names of operations defined over a type but also the types of the operations. This is important development information since one wants to know not only what is available but how to use it. However, the information is purely syntactic and provides no semantic guarantee. That is, one can guarantee that a named function exists over a certain type, but one cannot guarantee that the behaviour implied is actually delivered. This would require a formal approach to development that is beyond the scope of FAD.

FAD Notation

A signature is represented in FAD by a double-edged rectangle as shown in Figure 14. The notation was chosen since a permissive signature is in essence an *outerface* to a

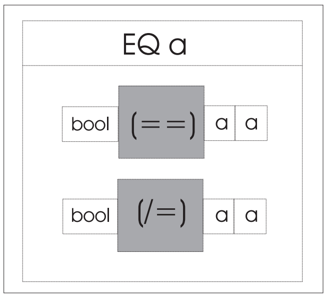

Figure 14: A Permissive signature

type (as opposed to an interface) and the notation mimics such a wrapping around a type. That is, a type with an extra layer of information. It encloses the signature's name which may be followed by the name of the instantiating type or type constructor, or a type variable. If the instantiating type is clear by the context of its use, then the type name can be left out. Section 5.4.4 describes type/signature associations.

Any operations of the signature may be added below the signature's name (either graphically or using the textual syntax `name : type`), separated by a horizontal line. One may elide a signature if it has a large number of operations or if the operations are presented elsewhere such as an inherited signature.

That completes the description of FAD's micro units. In order to model large systems one needs to be able to describe modular structures and their relationships. Thus FAD's modelling language includes a collection of *macro units* which are described in the following section.

5.3 FAD Macro Units

FAD's micro units and their various relationships deliver models of the functionality and data structures required of a system. The relationships are described in Section 5.4. They do not provide a means of describing the high level modular structure of a system. For this we require the macro units of FAD, which are modules, subsystems, exclusive signatures, projects, and files. In the following sections we present descriptions of each of these units accompanied by their graphical notation. The various macro unit and macro/micro unit relationships are described in Section 5.4. Figure 15 presents a 'Macro

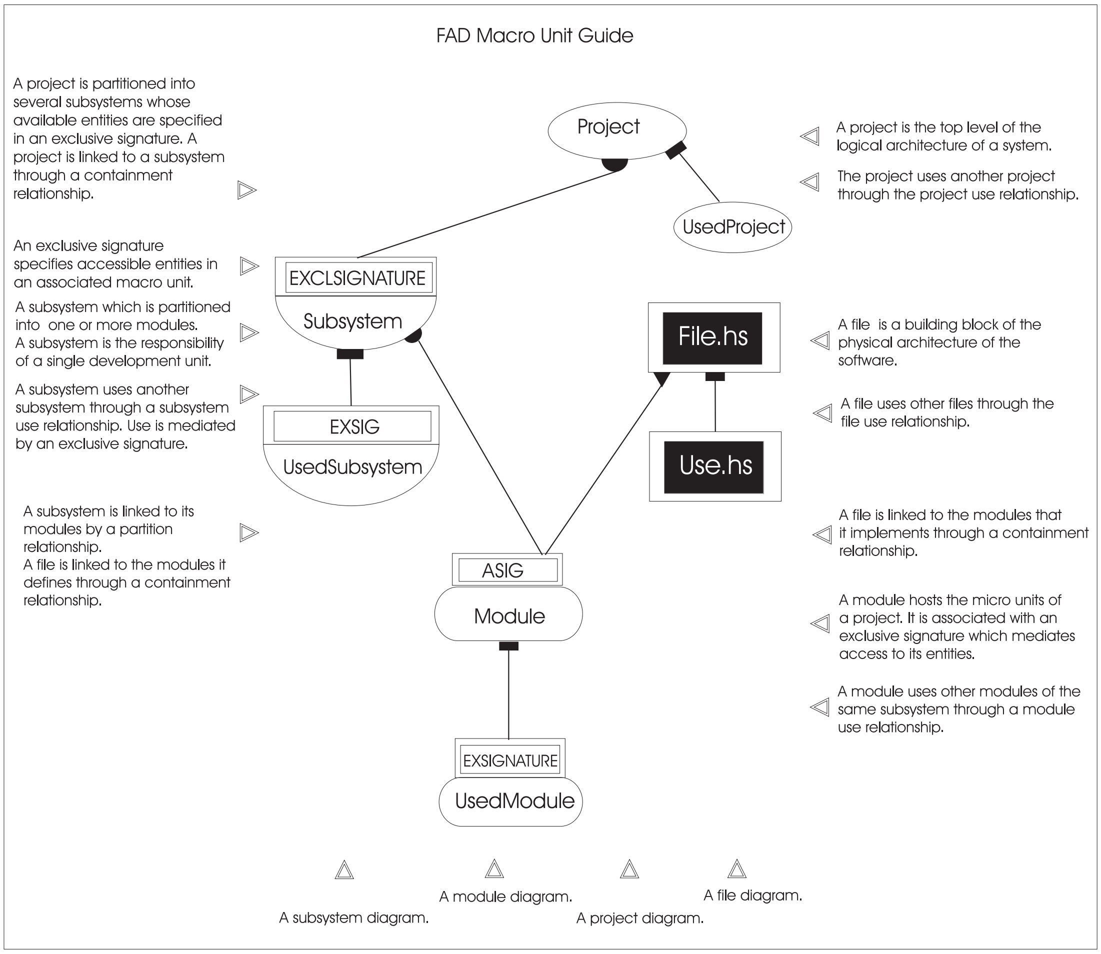

Figure 15: Macro Unit Guide

Unit Guide' which summarizes the macro units, their relationships and diagrams.

A project is the software system being developed. That is, it is the collection of micro units gathered in some hierarchical architecture to deliver the functionality required of an automated system. A project can be partitioned into a collection of linked subsystems (*subsystem architecture*), each of which can be further partitioned into several modules with various inter-dependencies (*module architecture*). That is, subsystems partition a project, which are themselves partitioned by modules. Each module is the host of the definitions of various micro units. Each subsystem can be used in other projects independently of the project for which it was originally developed. This is also true of modules. Therefore, there are several levels of reusability within a project. The project itself can become a component of a new project. A subsystem can be used in the development of a new project, and modules can be used independently in the development of subsystems of new projects.

5.3.1 Module

A module is an identified collection of micro units. In FAD, a project is partitioned into a collection of subsystems (which are described in the following section) and these are further partitioned into a collection of modules. Every type, function and permissive signature is declared in a module, which provides a medium for the development of a cohesive unit and in association with exclusive signatures, support for encapsulation and a mechanism for type abstraction. Every entity declared in a module is visible from every other entity declared in the same module. Entities declared in module **A** can use entities declared in module **B** if there is a *module use relationship* from **A** to **B**, and the required entity is specified in the mediating *exclusive signature*. The module use relationship is described in Section 5.4.10, and exclusive signatures in Section 5.3.3. If the two entities are declared in modules of different subsystems then the subsystems must be associated through a *subsystem use relationship* as described in Section 5.4.11.

Thus FAD supports modular program development based on information hiding through the use of modules, subsystems and their associated exclusive signatures. The methodology encourages the development of an architecture that maximises the cohesion of its units and minimises the coupling between the units. This is fully described in Chapter 7.

All modern functional programming languages support a modular approach to program development. Although there is some commonality in their approaches there are also some significant differences as described in Section 3.2.3.

The details of each module are described in a *Module Description Document* (MDD) as illustrated by the MDD for `TeamsMod`, the module which delivers the types and functions associated with football teams. This MDD is presented in Figure 16.

In each MDD, the project within which the module is defined is presented to the right of the header. The list in the body of the document presents:

- the module's unique name which begins with an upper case letter;

- the version of the module;

Module Description Document	Football
Name:	`TeamsMod`
Version:	19990711:1
Type(s):	`teams`, `matchTeams`
Permissive sig(s):	`TEAMSCON`, `MATCHTEAMSCON`
Function(s):	`addResultsToTeams:`
	`results -> teams -> teams`
	`addResultToTeams:`
	`result -> teams -> teams`
	`addTeams: matchTeams -> teams -> teams`
	`selectTeams:`
	`result -> teams -> matchTeams`
	`updatePerfs:`
	`matchTeams -> result -> matchTeams`
Modules used:	`ResultsMod : RESULTSSIG3`
Subsystem:	`FootballSS`
File:	
Description:	

This module hosts the type of football teams and its associated functions. It also hosts the type which represent the teams which played in a match. The type which represents a football team will be hosted in a separate module to decouple it from the teams type.

Figure 16: Module Description Document for the Module `TeamsMod`

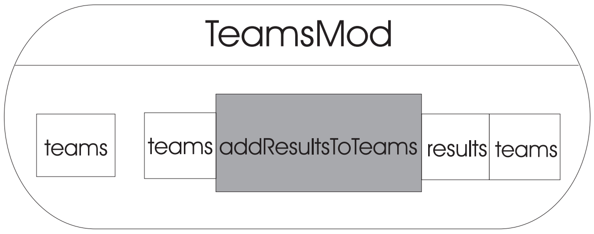

Figure 17: The Module `TeamsMod`

- the types, permissive signatures and functions declared in the module. Each function is accompanied by its type;
- the modules used by the module. In each case the module name is declared with the associated exclusive signature which mediates its use. The name of the module and signature are separated by a colon;
- the subsystem within which the module is declared;
- the file in which the module is implemented;
- a description of the module.

In common with FDDs and TDDs, the description document for a module will be updated to record iterative developments of the module.

FAD Notation

A module is represented in FAD by a semi-circular ended rectangle enclosing the module's name. Since a module supports encapsulation which can be defined *as in a capsule*, we have chosen a capsule-like notation. One can enclose any subset of the module's entities represented graphically or textually. The module `TeamsMod` is presented with one of its functions in Figure 17.

5.3.2 Subsystem

A subsystem is a collection of modules and exclusive signatures. That is, each module is declared in a subsystem along with any associated exclusive signatures. The rules regarding module/exclusive signature associations are described in Section 5.4.5. Each subsystem should be developed by a single development unit. Partitioning a project into a collection of subsystems supports an incremental approach to software development and provides a robust filing system for system entities. That is, each entity will be defined in a named module, which itself is part of a named subsystem.

An entity **EA** of a module **A** may use an entity **EB** of module **B** declared in the same subsystem, if there exists a module use relationship from **A** to **B** which is mediated by an exclusive signature in which **EB** is specified. If however, modules **A** and **B** are declared in the subsystems **SA** and **SB** then there must be a *subsystem use relationship* from **SA** to **SB** which is mediated by an exclusive signature in which **EB** is specified. We describe the subsystem/exclusive signature association in Section 5.4.6 and the subsystem use relationship in Section 5.4.11. Exclusive signatures are an important developmental aid in that they support the *principle of least commitment*, where one can delay detailed design until absolutely necessary. The rôle of exclusive signatures in development using the FAD methodology is described in Chapter 7.

Subsystems are not assigned a unique construct by any functional programming languages. However, they can be realised through the modular system of each language. For example, in SML a structure (which is a collection of declarations) can include other structures. Similarly one can use Haskell's module import mechanism to mimic the assignment of several modules to a single module, which then controls access to all the modules through a single interface. Thus a subsystem-based design can be supported by modern functional languages.

The details of each subsystem are described in a *Subsystem Description Document* (SSDD) as illustrated by the SSDD for the subsystem `FootballSS`, the subsystem which will deliver the problem domain functionality for the football system. That is, it will deliver through a collection of modules, the essential types specific to the football system, `teams`, `results` and so on, and the functions which implement the functionality required of any football league. This SSDD is presented in Figure 18.

In each SSDD, the project within which the subsystem is declared is presented to the right of the header. The list in the body of the document presents:

- the subsystem's unique name which begins with an upper-case letter and must not clash with any module or existing subsystem name;

Subsystem Description Document	Football
Name:	`FootballSS`
Version:	19990821:0
Module(s):	`TeamsMod : TEAMSSIG`
	`PlayersMod : PLAYERSSIG`
	`ResultsMod : RESULTSMODSIG1`
	`LeagueTableMod : LTSIG`
	`TeamMod`
	`ResultMod`
	`PlayerMod`
Exclusive Sigs:	`RESULTSSIG2`, `TEAMSIG`
	`PLAYERSIG`, `RESULTSIG`
Subsystems Used:	`GeneralSS : GENERALSIG`
Developed by:	
Description:	

This subsystem hosts the modules which are essential to the processing of football results. That is, the modules host the football related types and functions. The subsystem also includes the exclusive signatures which provide the interfaces to its modules.

Figure 18: Subsystem Description Document for the Subsystem `FootballSS`

- the version of the subsystem;

- the modules declared in the subsystem. Those modules that are associated with the subsystem via a partition relationship (described in Section 5.4.14) are presented with the exclusive signature which mediates the relationship. The exclusive signature makes explicit the module's entities that can be used by a client from another subsystem. That is, these are the only entities that can be specified in any exclusive signature associated with the subsystem. Modules which are only used by other modules of the subsystem are presented without an accompanying signature;

- the other exclusive signatures declared in the subsystem. These signatures are used to mediate interaction between the modules of the subsystem;

- the subsystem(s) used by the subsystem and the associated exclusive signatures which mediate access to their entities;

- a reference to the programming unit which is responsible for the development of the subsystem;

- a description of the subsystem.

The subsystem `FootballSS` hosts seven modules, and is dependent on a single subsystem `GeneralSS` that provides types and functions that are of general use, such as those typically declared in a language's standard environment.

FAD Notation

A subsystem is represented in FAD by a semi-ellipse enclosing the subsystem's name. This notation was chosen since a project is represented as an ellipse, and a subsystem is a part of a project. The modules declared in the subsystem can be presented textually below a horizontal line which delimits them from the subsystem's name. Alternatively one can present hosted modules through the *partition relationship* described in Section 5.4.14. We present the graphical notation for a subsystem in Figure 19.

Figure 19: A Basic Subsystem

5.3.3 Exclusive Signatures

The development of any large system requires the division of work among several development units. How one divides the work and the information provided to each development team, is essential to successful development. In FAD the unit of subdivision is the *subsystem* which was described in the previous section. The information regarding what is required of a subsystem, and how each can interact with other subsystems, is provided by *exclusive signatures*. They are also used to guide the development of modules. That is, during software development exclusive signatures play an essential rôle in specifying system requirements, and later in designing an implementable solution. Full details of the methodology and the techniques that develop exclusive signatures are given in Chapter 7.

An exclusive signature specifies a collection of micro units. These units are the only units visible to a client declared in another macro unit. A module or subsystem can only be used via an associated exclusive signature which declares the entities that are available for use. That is, an exclusive signature mediates access to an associated item. Module use and subsystem use are described in Sections 5.4.10 and 5.4.11 respectively.

Each signature entity is accompanied by its type specification. An exclusive signature can be associated with any module or subsystem which provides a binding for all of the signature's entities. This does not imply that the bindings are hosted by the associated macro unit, but that the unit is visible from the associated macro unit. Visibility of one micro unit from another is defined in Section 5.4.1. Module and subsystem associations with exclusive signatures are described in Sections 5.4.5 and 5.4.6 respectively.

Standard ML signatures, Miranda abstract type signatures, Clean definition modules and Haskell module export and import lists are thus supported through FAD's exclusive signatures. Some recent research has focused on using parameterised signatures to support a type-theoretic framework for modular programming [69]. FAD however has a clear distinction between the semantics of a (parameterised) permissive signature and that of a (non-parameterised) exclusive signature. A permissive signature presents the minimal functionality supported by its associated type(s), where an exclusive signature mediates access to the entities of an associated item. That is, a permissive signature specifies *at least this* where an exclusive signature specifies *only this*.

The implementation details regarding signature declaration and application are language-specific and are not a design issue. FAD provides a clear description of a design decision without imposing a particular implementation approach. The FAD description may present more information than that provided by an implementation language. Haskell, for example, presents (in a module's export or import list) the names of accessible entities without any type information (although this may be added to Haskell 2). In contrast, ML signatures and Clean definition modules provide type information alongside the entity names.

The details of each exclusive signature are presented in an *Exclusive Signature Description Document* (ESDD) as illustrated by the ESDD for `TEAMSSIG`, an interface to the module in which the type `teams` and associated types and functions are defined. This is presented in Figure 20.

The project within which the signature is defined is presented to the right of the header. The list in the body of the document presents:

- the unique name of the signature written in upper-case letters. The name must not clash with any (permissive or exclusive) existing signature name;
- the version of the signature;
- the subsystem in which the signature is declared. If the signature is associated with a subsystem then this will appear blank since it is declared in the project and not any of its subsystems;
- the types specified in the signature. If the type's constructor signature is not specified in the signature then the type is used as an abstract type. Section 6.4 provides full details of FAD's support for abstract types;

Exclusive Signature Description Document Football

Name:	`TEAMSSIG`
Version:	19990827:0
Subsystem:	`FootballSS`
Type(s):	`teams`
Permissive sig(s):	
Function(s):	`addResultsToTeams:`
	`results -> teams -> teams`
Inherited Sig(s):	
Description:	

This signature provides an interface to the module `TeamsMod` when used by its subsystem.

Figure 20: Exclusive Signature Description Document for `TEAMSSIG`

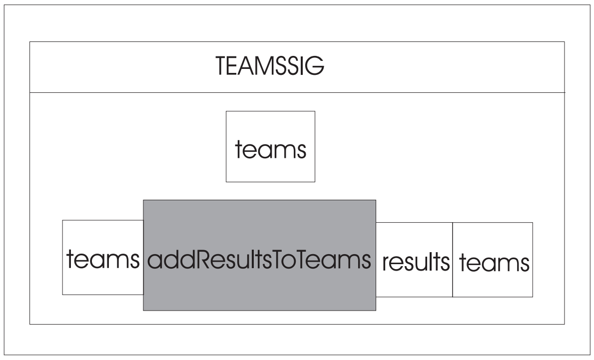

Figure 21: The Exclusive Signature `TEAMSSIG`

- the permissive signatures specified in the signature. Any constructor signatures will appear here;
- the functions specified in the signature with their type specifications;
- the signatures inherited by this signature. Signature inheritance is described in Section 5.4.7;
- a description of the signature.

FAD Notation

An exclusive signature has the same graphical notation as its permissive counterpart although its name will always appear by itself. The notation was chosen since an exclusive signature acts as an an interface to an associated macro unit and the notation mimics such a barrier to entry. This is illustrated with the exclusive signature `TEAMSSIG` presented in Figure 21.

5.3.4 Project

A system is developed as a project. A project is typically partitioned into several subsystems. Thus one declares subsystems and their associated exclusive signatures in a project. A project in no sense owns its subsystems. That is, any collection of the subsystems can be used in the development of another project. The only constraints on the use of a subsystem's entities are those imposed by an associated exclusive signature.

Project Description Document

Name:	`Football`
Subsystem(s):	`FootballSS : FOOTBALLSIG`
	`UISS : UISIG`
	`FileSS : FILESIG`
	`ParseSS : PARSESIG`
	`GeneralSS : GENERALSIG`
Exclusive Sigs:	
Projects Used:	
Development Units:	
Description:	

A project which implements an automated football results processing system.

Figure 22: Project Description Document for the Project `Football`

The details of each project are described in a *Project Description Document* (PDD) as illustrated by the PDD for `Football`, the football system project. This PDD is presented in Figure 22.

A PDD presents:

- the unique name of the project which begins with an upper-case letter and must not clash with any names of modules, subsystems or existing projects;
- the subsystems declared in the project. These are presented with the associated exclusive signature which mediates use of the subsystem's entities. That is, any other signature associated with a subsystem must provide a subset of the specifications declared in this signature;
- the other exclusive signatures declared in the project. These are used to mediate interaction between entities of the project's subsystems;

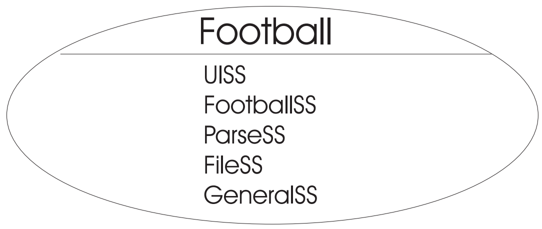

Figure 23: The Project `Football`

- the other projects used by the project;
- the development units assigned to the project, and the subsystem(s) for which they are responsible;
- a brief description of the project.

FAD Notation

A project is represented in FAD by an ellipse enclosing the project's name. An ellipse was chosen since it is nicely represents the *global* nature of a project. Below a delimiting horizontal line one can present the names of the project's subsystems. Alternatively these can be linked to the project using the partition relationship described in Section 5.4.14. The graphical representation of the project `Football` is illustrated in Figure 23.

5.3.5 File

Each project will be implemented as a collection of files. These may include standard environment file(s), library files, data files and files in which the project's modules are declared. That is, a file is a component of the system that delivers a part of an implemented project. Where the subsystem and module architecture provides a *logical model* of a system, the collection of files and their collaborations describe a *physical model* of the software which implements the system. Since files are units of implementation, their architecture is determined late in any development process.

Every module will be defined in a single file but a file could include the definition of several modules. A subsystem will normally be defined through several files, but every file will be associated with a single subsystem. Every exclusive signature will be defined in a single file although once again several could be defined in the same file.

File Description Document	Football
Name:	`Teams.hs`
Subsystem:	`FootballSS`
Module(s):	`TeamsMod`
Exclusive sig(s):	`TEAMSSIG`
Data hosted:	
Files used:	
Description:	
The implementation of the football teams module.	

Figure 24: File Description Document for the File `Teams.hs`

The details of each file are described in a File Description Document (FIDD) as illustrated in Figure 24 by the FIDD for `teams.hs`.

In each FIDD the project being implemented is presented to the right of the header. The list in the body of the document presents:

- the file's unique name which will be written in a manner consistent with the implementation language;
- the subsystem supported by the file;
- the module(s) implemented in the file;
- the exclusive signatures implemented in the file;
- the data hosted by the file. For example, the current record of the football teams;
- the files used in the implementation of the file. The *file use relationship* is described in Section 5.4.13;
- a description of the file.

Each modern functional programming language adopts its own conventions regarding the assignment of modules to files. In Haskell each module must be declared in a separate

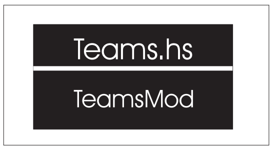

Figure 25: The File `Teams.hs`

file typically of the same name. The module definitions are normally accompanied by an export list of entities available to potential clients. SML imposes no such restriction, and thus multiple modules can be declared in a single file. Clean requires two files for any module which contains entities available to other modules, one to host the definitions and the other to declare the entities that are for export. In each case the file name must match the module name with the file extension signalling its use. That is, an implementation module file has the extension `icl` as opposed to `dcl` for a definition module file. Miranda has no language notation for a module, providing its support for modular programming directly through its files. A more detailed description of modular support in modern functional languages is presented in Chapter 3.

FAD Notation

A file is represented in FAD as a blackened rectangle with a white border. This looks similar to a filing cabinet with the names representing each drawer. The file name is written in the rectangle, which can also include the name(s) of the module(s) declared in the file. This is illustrated in Figure 25.

This concludes the description of FAD's micro and macro units. How they collaborate is described in the following section.

5.4 FAD Relationships and Associations

Various relationships and associations between the modelling language's units are supported by FAD. These include instantiation of a permissive signature by a type, module/exclusive signature association and several 'use relationships'. In this section we will describe the syntax and semantics of each relationship. We will illustrate each with an example from the case study.

5.4.1 Argument of a Function

Since all data flow is explicit in a pure functional program, and most modern functional languages are strongly typed, the argument and result type(s) of a function play an important role both in guiding development of software and in recording the characteristics of a program. Although a functional programmer is not required to specify the types of functions, as is the case in statically-typed OO languages, as a development tool it is extremely beneficial and is therefore encouraged by FAD. The relationship between a function and its argument (and result types) is a *use relationship* since the function uses values of the argument type(s) to create values of the result type.

All argument types must be *visible from* their associated function. The visiblity rules are the same for all micro units. That is, micro unit `B` is visible from micro unit `A` if and only if precisely one of the following is true:

- **A** and **B** are hosted by the same module;
- **B** is hosted by a module **BMod** in the same subsystem as the module **AMod** which hosts **A**. There is either a module use relationship from **AMod** to **BMod** with **B** specified in the mediating exclusive signature, or there is a path from **AMod** to **BMod** via one or more intermediate modules where each module use relationship linking the modules is mediated by an exclusive signature that specifies **B**;
- **B** is hosted by a module **BMod** hosted by a subsystem **BS** which is used by the subsystem which hosts the module in which **A** is declared. **B** must be specified in the exclusive signature which mediates use of the subsystem, and in the exclusive signature which mediates the partition relationship between the subsystem **BS** and **BMod** or a module which is linked to **BMod** via a path as described in the case above. This is illustrated in Figure 91 where to aid readibility we have limited the specifications presented in the exclusive signatures to those required for the example.

The module use relationship, subsystem use relationship, partition relationship, module/exclusive signature association and subsystem/exclusive signature association are described in Sections 5.4.10, 5.4.11, 5.4.14, 5.4.5 and 5.4.6 respectively.

Polymorphic functions are restricted in their application to types that are visible. Constrained polymorphic functions are dependent on the permissive signature which declares the constraint. They are restricted in their application to types that are visible and instantiate the permissive signature as described in Section 5.4.4. This implies that software must be designed in such a way that a function has access, maybe only in an abstract sense, to its argument type(s).

Higher-order functions with functional arguments imply a dependency between the higher-order function and any actual functional argument. This is described in Section 6.9.

FAD Notation

A function argument type is represented in FAD through the juxtaposition of the type to the right of the function as illustrated in Figure 26. The type boxes are external to the function box since it: conforms to the juxtaposition-based syntax between a function and its arguments found in most functional languages; it avoids potentially messy nested notation for the representation of permissive signature/type associations as described in Section 5.4.4; and, it simplifies the representation of functions with functional arguments.

To support modular development, one can annotate the type notation to indicate whether the function and type are declared in the same subsystem or if they are declared in the same module. The default notation represents an intra-module relationship. That is the function and type are declared in the same modules. An inter-subsystem relationship is indicated by a broken vertical line in the type box at the function end of the link. An intra-subsytem, inter-module relationship is indicated by a solid vertical line in the type box at the function end of the link.

The function `checkResult` which checks the acceptability of a result against existing results and the collection of football teams, is declared in the module `ResultMod` of the subsystem `FootballSS`. It takes three arguments. The first is of type `result` which is declared in the same module. The second and third of types `results` and `teams` are declared in the modules `ResultsMod` and `TeamsMod` of the same subsystem. The result type `bool` is a general-purpose type that is declared in a module of the subsystem `GeneralSS`.

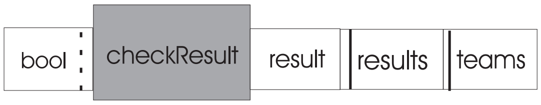

Figure 26: A Function and its Type with Modular Annotations

5.4.2 Result of a Function

A function is dependent both on its argument type(s) and result type. Therefore, the visibility rules described in the previous section equally apply to a function and its result type. Hence a function has a use relationship with its result type and the same design implications apply as those stated in Section 5.4.1.

FAD Notation

A result type is represented in FAD through juxtaposing the type box to the left of its function box. That is, a type to the left of a function box is the result type of the function. The reasons for this notation are as described for an argument type.

A function/result type association is also illustrated in Figure 26.

5.4.3 Curried Functions

All modern functional languages afford the developer a choice of designs for multiple argument functions. The first form, which is also common to non-functional languages, is to present the arguments in a tuple. The second form delivers the arguments one at a time and is known as the *curried* form. The benefits of *currying* were described in Section 3.1.4.

FAD Notation

Curried functions are represented through juxtaposing the first type box to the right of the function box, and then each further type box to the right of the previous type box.

In Figure 27 we present FAD notation for the curried function `addResultToPlayers` which in Haskell has the following specification.

```
addResultToPlayers :: Result -> Players -> Players
```

Figure 27: The Curried Function `addResultToPlayers`

Figure 28: Partial Application of the Function `select`

New functions can be statically or dynamically created through the partial application of the function to an incomplete set of argument values. FAD represents partial application by replacing a type with a value of a type as illustrated in Figure 28 where the function `select` is applied to a functional value `selectNameAndData`. `select` is a higher-order function which retrieves data from a collection of values by applying its first argument to each element in its second argument.

5.4.4 Type/Permissive Signature Association

A permissive signature provides the minimum functionality supported by any associated type. There are two forms of association that FAD supports. The first is that between type variables and a permissive signature which we call the *type/permissive signature contract association*. A permissive signature restricts the type which can be bound to the type variable(s) to those that provide bindings for each of the signature's operations. These types are linked to the permissive signature through the second form of association that we call the *type/permissive signature instantiation association*. The type constructors of any type(s) that instantiate a permissive signature must have the same *kind* as the signature's parameter(s). Type instantiation of a signature implies that bindings exist for the operations of the signature defined over the type.

FAD Notation

The association between a permissive signature and a type (or type variable) is represented through juxtaposing the two. Juxtaposition was chosen since a permissive

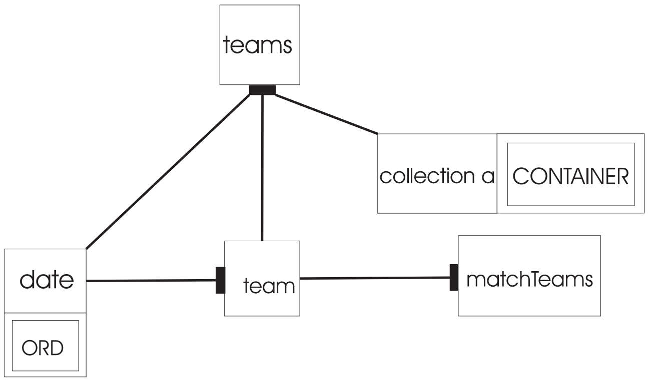

Figure 29: Type Dependency Diagram for the Type `teams` with Signature Instantiation

signature is adding an extra layer of information to a type. This is illustrate in Figure 29 where we present part of the *type dependency diagram* for the type `teams`. We will return to this diagram to illustrate other relationships but for now we focus on the instantiation of the permissive signature `ORD` by the type `date`, and the permissive signature `CONTAINER` by the type `collection a`. In both cases one could represent the instantiation simply through the signature notation with the entry `EQ date` and `CONTAINER collection` respectively. Here one simply presents the type constructor name (without any parameters) after the permissive signature name.

When more than one signature is instantiated by a type this can be represented either by juxtaposing the signatures, or juxtaposing each signature with the type. In addition one can represent multiple instantiations of a single signature by juxtaposing the signature with each type as in Figure 30.

Thus the types `int`, `bool`, `char`, and `float` all instantiate the signature `EQ`.

Instantiation of a multiple parameter permissive signature is represented by enclosing the instantiating types inside a type box juxtaposed with the signature. We illustrate this in Figure 30 with a FAD representation of an example similar to one described in [102]. In [102] the example refers to a multiple parameter type class `Collection` with two parameters of kind `* -> *` and `*`. The second parameter enables constraints to be applied to the type variable which represents the elements of a collection type. We have called the permissive signature `SET`.

One can include type/permissive signature associations in the description of a function. The methodology encourages such associations in a function description since they

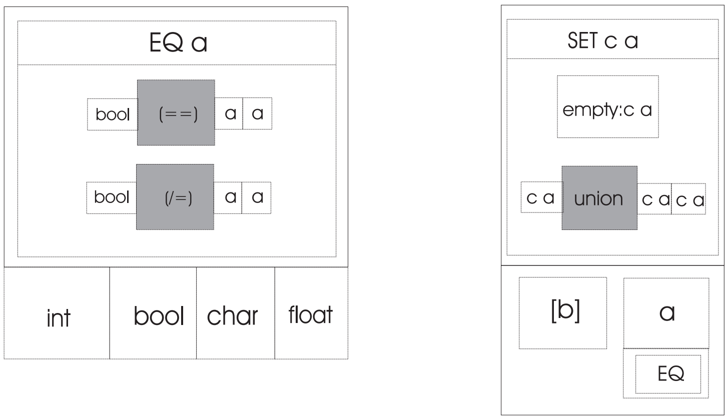

Figure 30: Type Instantiation of a Signature

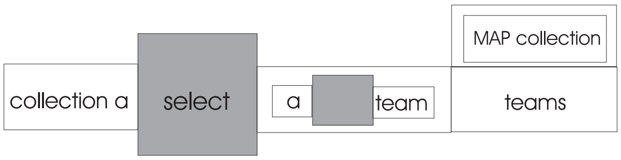

Figure 31: Type Constructor/Signature Association

provide information regarding the potential for higher-order and constrained polymorphic functions. They also provide a key for storing a function in the data dictionary. FAD's techniques for developing higher-order, and overloaded or polymorphic functions is described in Sections 7.3.3 and 7.3.2. The data dictionary is presented in Chapter 8.

We illustrate with an example from the case study. The function `select` was first described in Section 5.4.3 to illustrate partial application. The description of the function in Figure 31 has been updated with the association of the permissive signature `MAP` with the type `collection a`, which is used by the type teams to construct values of the type. This indicates that the function `select` requires 'mapping' behaviour over its second argument. That is, it needs to apply a function to each of the elements in a collection. The type description of the type `collection a` will need to be updated accordingly unless the instantiation has already been declared.

Function overloading is not supported by all modern functional programming languages. Miranda only provides overloading for the built-in comparison operators and the function `show` which converts a value to its printable form as a string. SML allows function identifier reuse through module name qualification but not function overloading. Haskell and Clean both provide first and higher-order overloading through type and constructor classes [66]. The class presents the signature supported by any instantiating type. A class can therefore implement a permissive signature.

Although support for function overloading is not provided by all functional languages, the design benefits of making explicit the behaviour required by a type, or the behaviour defined over a type is invaluable during development. Permissive signatures and their associations can be modelled either directly or indirectly in any modern functional language.

5.4.5 Module/Exclusive Signature Association

In Section 5.3.1 we presented a brief overview of the support within functional programming for modular programming. When designing a system it is important to be able to separate the implementation of a module's entities from its interface to the outside world so that the effect of any implementation changes are localised. FAD supports this approach both notationally and in its methodology described in the following chapter.

FAD provides modules in which micro units are defined, and exclusive signatures that specify an interface to a module. A module/exclusive signature association specifies the entities of a module which are available to a client module which is linked to the module via a *module use relationship.* Each entity specified in the signature is either declared in the associated module, or in a module which is connected to the associated module by a path of module use relationships and is specified in each mediating exclusive signature. Thus one can associate an exclusive signature with any module which can provide a binding for each entity specified in the signature, where the binding may be provided by entities declared in the module, declared in modules (and specified in the associated exclusive signature) used by the module, or declared in a module used by a used module (and specified in the associated exclusive signatures) and so on. The module use relationship is described in Section 5.4.10.

Each module will be associated with at least one exclusive signature, but could be associated with several signatures. Each signature will present an interface to the module for a particular client. For example, module **A** may require access to the types declared in module **B** and require knowledge of how they are constructed. Module **C**

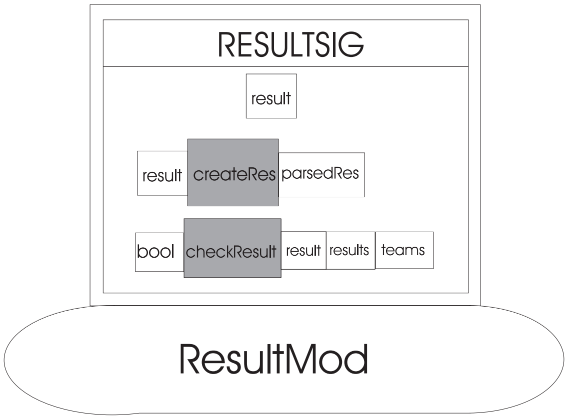

Figure 32: Module/Exclusive Signature Association for the Module `ResMod`

simply requires access to the types of module **B** and some operations over the types. The exclusive signature associated with module **B** and used by module **A** will include the types of **B** and their associated constructor signature, often referred to as a *transparent signature*. In contrast, the signature used by module **C** includes the types without their constructor signatures (abstract data types) and the required operations. Multiple interfaces to a single module are supported by most modern functional languages.

FAD Notation

A module/exclusive signature association is represented through juxtaposing an exclusive signature with a module. We chose this notation since an exclusive signature provides an *interface* to the macro unit to which it is juxtaposed. In Figure 32 each entity of the signature `RESULTSIG` is declared in the module `ResultMod`. Entities not specified in the signature may also be declared in the module. They are not however visible to external clients.

Hence FAD supports and encourages the separation of a module definition from its interface, and encourages the explicit statement of the functionality availed by a module through its associated signature(s). This allows the developer to describe the collaboration between modules at the interface level before focusing on the internal implementation details of each module.

5.4.6 Subsystem/Exclusive Signature Association

In Section 5.3.2 we described subsystems and how they can be used during the development of a system. Subsystems provide a mechanism for managing large projects through hosting a collection of modules with some common purpose. The subsystem/exclusive signature association mirrors the module/exclusive signature association described in Section 5.4.5.

Every entity specified in an exclusive signature associated with a subsystem must also be specified in the exclusive signature which mediates use of a contained module's entities through a *partition relationship*, or in an exclusive signature which mediates use of another subsystem via a *subsystem use relationship*. We describe the subsystem use relationship in Section 5.4.11 and the subsystem/module partition relationship in Section 5.4.14.

During development the design of module interfaces is guided by the usage requirements of their host subsystem and not vice versa. That is, subsystem use drives the development of its modules and associated signatures. Full details of this process are presented in the following chapter.

FAD Notation

The subsystem/exclusive signature association in common with the module/exclusive signature association is represented in FAD by the juxtaposition of an exclusive signature with a subsystem. This notation was chosen for the same reasons presented in Section 5.4.5. This is illustrated in Figure 33 where the user interface subsystem `UISS` is associated with the exclusive signature `UISIG` which is presented in an elided form.

The signature declares a collection of I/O functions available for use. We describe in Section 7.2.1 some issues regarding the representation of impure actions within the purity of FAD.

5.4.7 Signature Inheritance Relationship

A signature can adopt the entities specified in another signature, through the transitive *signature inheritance relationship*. The only mechanism for respecifying an entity in a new signature is through inheriting its specification from an existing signature.

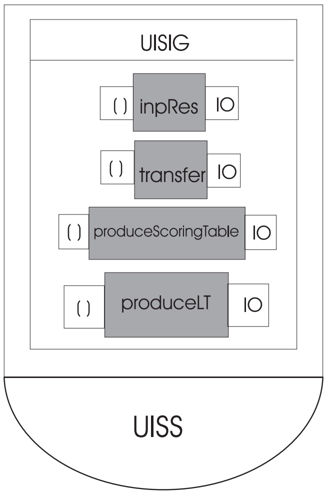

Figure 33: Subsystem/Exclusive Signature Association for the Subsystem `UISS`

A signature can only inherit from one or more signatures of the same form. That is an exclusive signature can only inherit from other exclusive signatures. Permissive signatures are restricted to inheritance of other permissive signatures where they have matching parameter *kinds*. A signature can only inherit from a signature that is visible. That is, if one wants a signature to inherit from another signature then they must either be declared in the same macro unit (the only possibility for exclusive signatures since they do not appear in other interfaces), or are declared in the appropriate interface(s). Since a permissive signature may be instantiated by several unrelated types they should be as visible as possible. For example, in the case study all permissive signatures are declared in the subsystem `GeneralSS` and specified in the mediating exclusive signature `GENERALSIG`. This subsystem is used by all other subsystems of the project.

In functional languages that support type and constructor classes, inheritance is a common mechanism for constructing new classes. For example, in Haskell 98 [100] several of the built-in classes such as `Eq` and `Ord` are related through inheritance.

FAD Notation

The signature inheritance relationship is represented by an arrow between two signatures, pointing towards the bequeathing signature and from the inheriting signature. Parameter names should be supplied when needed for clarification. For example, if a

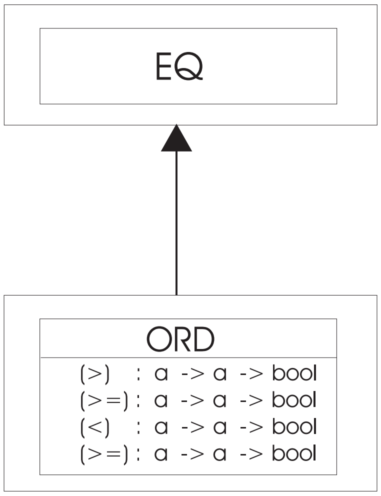

Figure 34: Signature Inheritance Relationship between `EQ` and `ORD`

multiple parameter signature inherits from a single parameter signature, one should use a consistent name for the related parameters in the two signatures.

The graphical notation is the reverse of that adopted in the Haskell 98 Report [100]. We argue that this is a more natural representation since the direction of the arrow reflects the fact that an inherited signature is implied by an inheriting signature. That is, if a type instantiates a signature `A` which inherits from signature `B` then it also instantiates signature `B`. A similar argument can be made for modules or subsystems and their associated signatures.

Extensible algebraic types have recently been mooted as a means of supporting subtyping within functional languages [107]. FAD supports them through the signature inheritance relationship between constructor signatures. As yet modern functional languages do not support extensible algebraic types.

We illustrate signature inheritance in Figure 34, where the permissive signature `ORD` inherits from the permissive signature `EQ`. The new operations specified in `ORD` are presented below its name.

In the following section we describe the various *use relationships* between units of the same form.

5.4.8 Type Use Relationship

A type can be defined in terms of one or more existing types. FAD's non-transitive *type use relationship* declares a unidirectional dependency from the using type to the used type(s). The using type could be an alias for the used type or could be a *composite type* whose values are constructed using values of the used type(s). For design purposes it is important to make explicit these dependencies since they will influence the architecture of the system. A type may only use a type that is visible.

A type is visible from another type if one of the cases for visibility presented in Section 5.4.1 is true. The relationship is non-transitive since the type **t1** could be visible from the type **t2** which is visible from the type **t3**. However the type **t1** may not be visible from **t3**. In a modular design in which a minimum of coupling between modules is practised, one would expect and encourage these patterns of design. A constrained parameterised type requires an association between a type variable and at least one permissive signature. The permissive signature(s) must be visible from the type, which will always be the case if one practises a design approach where all permissive signatures are visible from all entities.

FAD Notation

The type use relationship is represented by a link from the user type to the used type or an associated permissive signature. The link is connected to the using type by a filled-in rectangle. This notation was chosen because we required a simple (and reusable) notation that made clear the direction of usage. We use this same notation for all use relationships between units of the same form. In support of modular development the use relationships may reflect whether the entities at each end are declared in the same subsystem and also if they are declared in the same module. A broken line link indicates an inter-subsystem relationship; a thin line link indicates an intra-subsystem but inter-module relationship and a thick line link an intra-module relationship. The thin line link is used by default and will be updated if necessary.

A sum type can be modelled by annotating the use relationship with comma delimited natural numbers, to indicate which types and type constructors are used by each element of the sum. We need an annotation that supports more than one number since

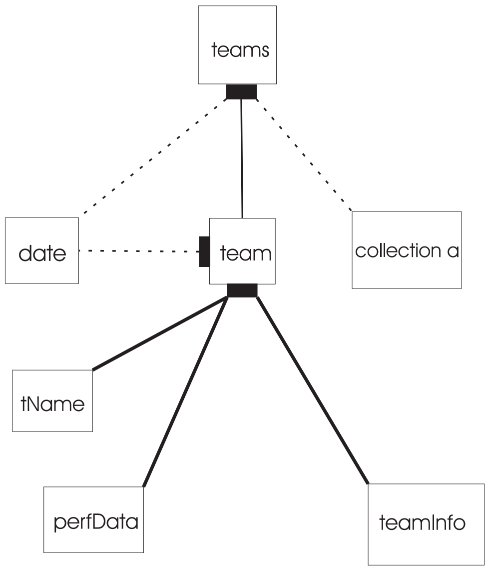

Figure 35: Type Dependency Diagram for the Type `teams`

some types will be used in more than one element of the sum.

We illustrate the type use relationship in Figure 35 with a *type dependency diagram* for the type `teams`. Type dependency diagrams present a data-centric view of a system or part of a system. The type `teams` uses three types, `date`, `team`, and `collection a` in the construction of its values. The types `teams` and `team` are both declared in modules of the subsystem `FootballSS`. The types `date` and `collection a` are defined in the subsystem `GeneralSS`. The type `team` uses the types `tName`, `perfData` and `teamInfo` which are all declared in the module `TeamMod`.

5.4.9 Function Use Relationship

Functional programmers are encouraged to design programs that are both 'modular-in-the-large' and 'modular-in-the-small'. FAD's macro units and macro unit relationships support the first form of modularity. The *function use relationship* supports the latter through the development of designs built on small functions with a clear single purpose.

FAD's non-transitive *function use relationship* declares a unidirectional dependency from a using function to a non-argument used function. The same visibility rules apply for used functions as for used types.

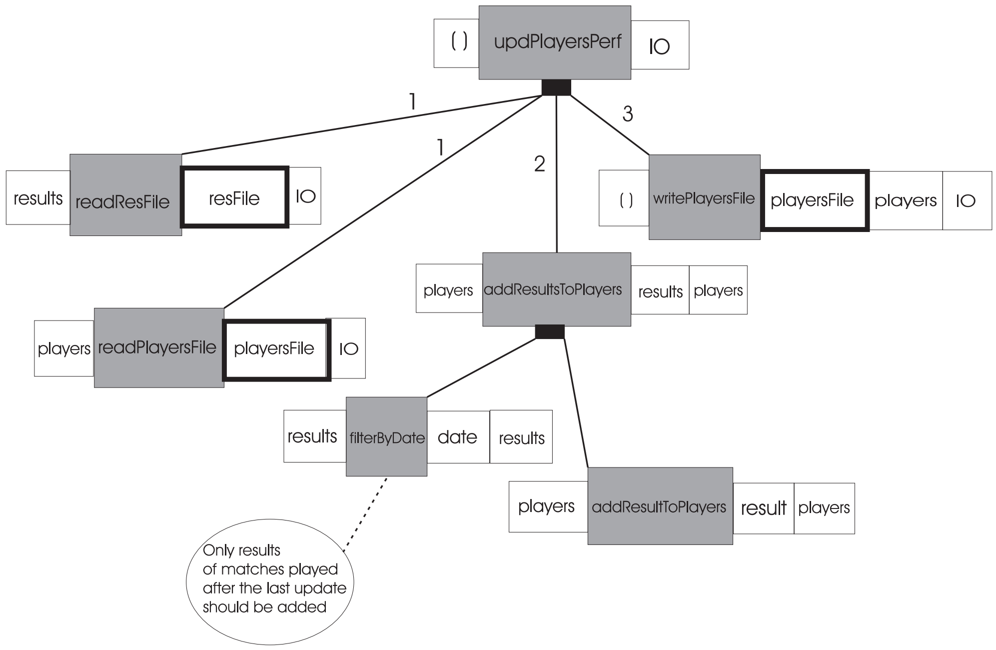

Figure 36: Function Dependency Diagram for the Function `updPlayersPerf`

FAD Notation

The functions used in the body of a function are linked to the using function through the same uses notation as for type use. This is illustrated in Figure 36 with the *function dependency diagram* for the function `updPlayersPerf`.

The I/O function `updPlayersPerf` uses the file I/O functions `readPlayersFile`, `readResFile`, and `writePlayersFile`. It also uses the function `addResultsToPlayers` which in turn uses the functions `filterByDate` and `addResultsToPlayers`.

One can annotate function use relationships to indicate sequentiality of used function application and conditional behaviour. One can also use annotation to indicate nested sequentiality. We first describe non-nested sequential annotation. Each use relationship link is annotated with a natural number that indicates the order of application of the functions. A function with a link indexed with a natural number **n** will be applied in advance of all functions with a link whose index is greater than **n** and after any with an index less than **n**. Since functions can exhibit both sequential and non-sequential behaviour, those functions with identical indexes require no mutually sequential application. If the use relationship links have no annotations then one can assume that no sequentiality of application of the functions is required.

Nested sequential behaviour is represented through qualified indexes. That is, the index is written by postfixing the index of the link to the using function, with a full stop followed by a natural number index. This indexing can be repeated to any level of dependency, although we would encourage models which have several levels of dependency to be represented using a collection of diagrams as is common when using data flow diagrams. That is, each function with significant dependency requirements should be described in a separate diagram.

Functions with conditional behaviour will require sequential behaviour for the determination of which case is true, and the evaluation of the associated expression. The implementation of the conditional function could be as a collection of guards or as a conditional expression. These details are left to the software implementers and may reflect the idiosyncrasies of a particular implementation language.

A conditional function is best represented using a separate diagram for each case. A condition can be represented as a function that returns a Boolean value. Success can be represented by the value `True` in the result rectangle and failure by the value `False`. A function with more than two cases will have more than one condition function. We illustrate in Figures 37(a) and 37(b) the FAD diagrams that model the function `condFun`. If the predicate function `predFun`, when applied to the inputted integer returns `True`, `condFun` uses the function `fun1`. Otherwise it uses the function `fun2`.

```
condFun :: Int -> Int
condFun i
        | predFun i     = fun1 i
        | otherwise     = fun2 i
```

Thus one can use annotations to aid the reading of multiple diagrams that represent the model of a function with conditional behaviour. In Figure 37(a) we represent the case where the first condition is satisfied. The annotations are simply those for sequentiality. Figure 37(b) models failure of the first condition and success of the second. The annotation to the function `fun2` is extended with the letter `a` to indicate that this is an alternative to the model in Figure 37(a).

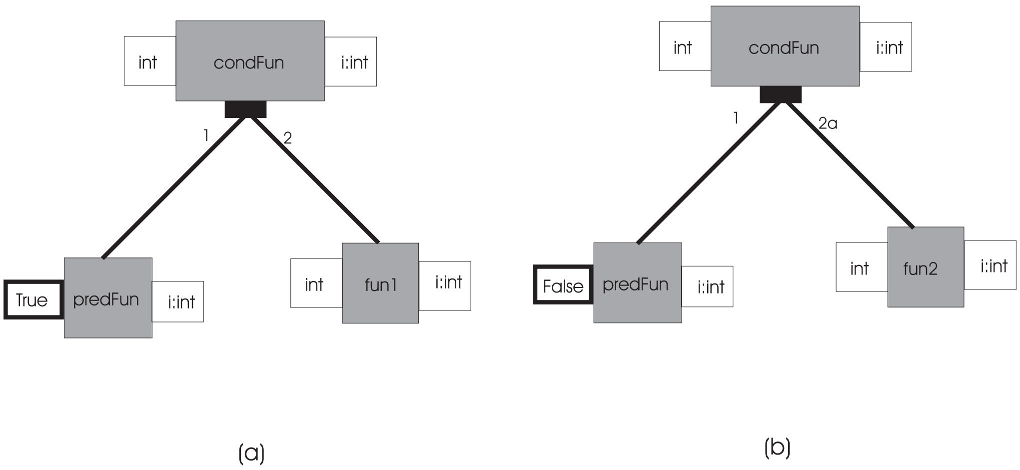

Figure 37: Conditional Function Diagrams

One can add comments to any FAD diagram through enclosing the commentary in a circle and attaching it to the relevant item through a broken line as illustrated in Figure 36.

5.4.10 Module Use Relationship

Hosting functions and types in modules aids the management of software development and if practised effectively will minimise the scope of any changes to the software. One should develop cohesive modules which have a minimal but explicit coupling with other modules. We describe in the following chapter how FAD's methodology both aids and encourages the development of modular designs where information hiding is the dominant criterion. In this section we describe how entities declared in one module can use entities declared in another module of the same subsystem. The following section describes a similar relationship between subsystems.

FAD supports inter-module development through its *module use relationship*. This is a non-transitive, unidirectional relationship between two modules mediated by an exclusive signature associated with the used module. Entities in one module may make use of entities declared in another module of the same subsystem if and only if there is a module use relationship from the client module to the used module. The entities available for use are those specified in the associated exclusive signature.

Module use is only supported between modules of the same subsystem. Entities

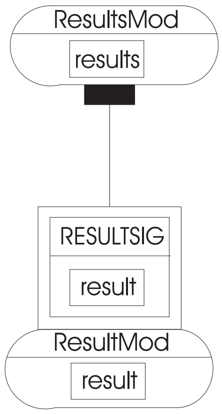

Figure 38: A Module Diagram

declared in modules of different subsystems of a project require a *subsystem use relationship* from the client subsystem to the used subsystem. This is described in the following section.

FAD Notation

FAD uses the same graphical notation for module use as for type and function use except one only uses the inter-subsystem and intra-subsystem versions of the notation. That is, a module use relationship is a rectangle-ended link between the client module and the used module, although it must be linked to an exclusive signature associated with the used module.

We illustrate module use in the *module diagram* presented in Figure 38. The module `ResultsMod` hosts the type `results`, which is a collection of values of type `result`. That is, `results` uses `result`. The type `result` is hosted by the module `ResultMod` and is specified in the associated exclusive signature `RESULTSIG`.

5.4.11 Subsystem Use Relationship

FAD not only provides modules to support the management of the software development process but also subsystems that host a collection of modules. One can make the same arguments for a sensible subsystem architecture as stated for the module architecture in the previous section. FAD supports inter-subsystem development through its *subsystem use relationship*, a non-transitive, unidirectional link between two subsystems.

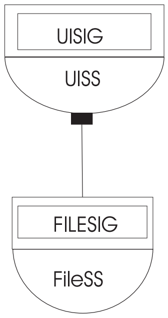

Figure 39: A Subsystem Diagram

A subsystem use relationship indicates that the entities of the client subsystem may be able to use entities declared in the used subsystem. A subsystem may only be used via an associated exclusive signature that specifies entities that are available for use.

The subsystem use relationship supports the dependency of a micro unit declared in a module of one subsystem on a micro unit declared in a module of another subsystem. Intra-subsystem dependency is supported by the module use relationship described in the previous section. That is, if a function declared in a module of one subsystem needs access to a function declared in a module of another subsystem then this is modelled in FAD through a subsystem use relationship between the relevant subsystems.

FAD Notation

The notation used in FAD is the same as the default use relationship notation for the type use, function use, and module use relationships. We illustrate in Figure 39 with a *subsystem diagram* from the case study.

The subsystem `UISS` that hosts the modules which implement user interface types and functions, is linked to the subsystem `FileSS` in which the file-handling functionality is supported. Various text-based I/O functions declared in modules of `UISS` depend on functions that write to files or read from files. These are declared in modules of the subsystem `FileSS`.

5.4.12 Project Use Relationship

A project can make use of another project through FAD's non-transitive, unidirectional *project use relationship*. Alternatively a project can use individual subsystems of another project, or develop new subsystems from the modules declared in another project. That is, although a project is partitioned into subsystems that themselves are further partitioned into modules, the architecture is project-specific. A new project can reuse an existing project with its declared architecture, or one or more of an existing project's subsystems with their declared architecture, or one or more modules developed for an existing project. In summary, subsystems are independent of the project for which they were originally developed. Modules are also independent of the subsystems for which they were originally developed. They can therefore be reconfigured to support a new project, or be used collectively as a component of a larger project.

FAD Notation

The notation for the project use relationship between two projects is identical to that for the subsystem use relationship, except there is no associated exclusive signature.

5.4.13 File Use Relationship

In Section 5.3.5 we described how software is implemented as a collection of files. The file architecture will depend both on the software design and the idiosyncrasies of an implementation language. For example, Clean requires each module to be declared in a separate implementation file with a single associated definition file that declares the interface to the implemented module. Thus a module/signature association will be delivered as two files linked by a use relationship. FAD's non-transitive, unidirectional *file use relationship*, declares a dependency between two files. That is, the client file hosts entities that are dependent on entities hosted by the used file. Access rights are determined at the logical level, subsystems, modules and so on and not at the physical level. Therefore, accessibility will be dependent on the logical architecture of the system. A system's file architecture is presented in a collection of *file diagrams* which are simply files linked by file use relationships.

FAD Notation

The file use relationship has the same notation as the default notation for all other FAD use relationships. Of course there are no exclusive signatures mediating access.

5.4.14 Partition Relationship

A FAD project is partitioned into one or more subsystems which are themselves partitioned into one or more modules. Each module hosts one or more micro units. These relationships are modelled in FAD as the transitive *partition relationships*. Thus a partition relationship links either a project with a subsystem or a subsystem with a module.

FAD Notation

A partition relationship is a filled semi-circle ended link from the partitioned macro unit to a partition element. This notation was also chosen for its simplicity. The semi-circle end emphasises that it is a whole/part relationship, where a semi-circle is a part of a circle. This relationship is illustrated in Figure 40 where the project `Football`'s partition includes the subsystem `FootballSS` that itself includes the module `ResultsMod`. If the partition element is associated with an exclusive signature this signature specifies the element's entities that can be included in an associated signature of the partitioned unit. This only applies to the subsystem/module partition.

5.4.15 Containment Relationship

A file contains one or more logical units. This implies that the unit is defined in the file. Of course, more than one file could implement the same unit and possibly in different languages. A file is linked to a contained unit through the *containment relationship*.

FAD Notation

A containment relationship is a filled triangle ended link from a file to a unit defined in the file. This notation was chosen for its simplicity. The triangle end was chosen to discriminate this relationship from the various use relationships and partition relationships. This relationship is illustrated in Figure 40 where the module `ResultsMod` and

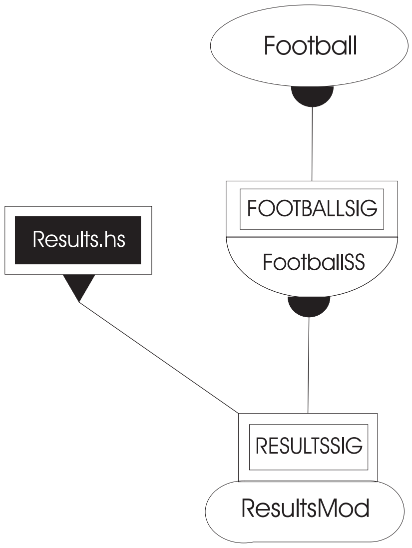

Figure 40: FAD's Partition and Containment Relationships

its associated signature are implemented in the file `Results.hs`.

5.4.16 FAD Comments

One can add comments to FAD diagrams. These can be attached to any FAD unit or relationship. They are used to add detail to a particular unit or relationship.

FAD Notation

FAD comments are presented inside a circle that is attached to the item for which the comment is made via a broken line. This notation was chosen since it looks like a 'callout', which is often used to relate text to an item on a picture or a slide. This is illustrated in Figure 41.

5.5 Summary

This chapter provided a description of the elements, syntax and semantics of the modelling language of FAD. There are three micro units, types, functions and permissive signatures and five macro units, projects, subsystems, modules, exclusive signatures and files. Various associations and relationships are supported between items of the same unit, and between items of different units.

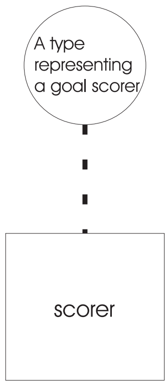

Figure 41: FAD Comment

The modelling language supports a range of diagrams that provide various views of a system. A function dependency diagram is a collection of functions linked by function use relationships. They model the functional requirements of a system and can include modular architecture information. A type dependency diagram is a collection of types and type use relationships. They present a static view of a system, and can also include modular architecture information. Project, subsystem and module diagrams model the various levels of a system architecture. A file diagram describes the physical architecture of an implemented system.

In the following chapter we illustrate how common designs used in functional programming can be modelled using this modelling language. In Chapter 7 we describe the methodology of FAD. It uses the elements presented in this chapter to develop models of a system.

Chapter 6

FAD Functional Designs

In Chapter 3 we described the main features of the functional programming paradigm and how they influence software development within the paradigm. Various designs are commonly used such as higher-order functions and algebraic types. In the previous chapter we described the modelling language of FAD. In this chapter we describe the modelling of common functional programming designs in FAD's modelling language. Since the language has been developed specifically to model functional programs, the designs should be natural to model. In practice however, one should not be looking to model particular designs but to model a problem, which can be iteratively developed to a model of an implementable design. Each design will be illustrated by an example and accompanied by a graphical representation of the FAD model.

6.1 Tuple Types

Tuple types are composite types with a special syntax in all modern functional languages, a parenthesis enclosed, comma-delimited collection of types. Values of the type are similarly represented with values replacing the types. Elements of a tuple value can be selected through pattern matching.

FAD Model

FAD represents a tuple type as a type that uses the tuple component types, and is associated with a constructor signature that specifies the relevant tuple-forming constructor.

We illustrate in Figure 42 with the model of the following pair type:

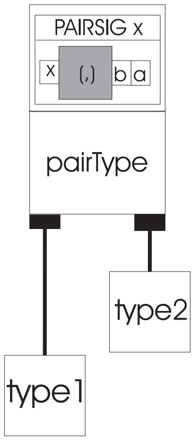

Figure 42: A FAD tuple type model

The pair

```
type pairType = (type1, type2)
```

6.2 Records

A record is similar to a tuple with the additional property of element selection through a *field name*. That is, a record is a tuple with named fields. For example, the record `aRec` (written in Hugs98 running in Hugs mode) has two fields, `a` of type `Int` and containing the number `3`, and `b` of type `Bool` and containing the Boolean value `False`.

```
aRec = (a = 3::Int, b = False)
```

Each record is accompanied by a set of selector functions - one for each field of the record. For example, the value held in field `a` can be inspected as follows:

```
#a aRec
```

Most modern functional languages support records. Hugs supports a flexible system of *extensible records* or "Trex" [68], the name reflecting the incremental building of the records. Clean and SML also support records but both are more restrictive in their use than Hugs. For example, in both these languages functions can only be defined over complete records.

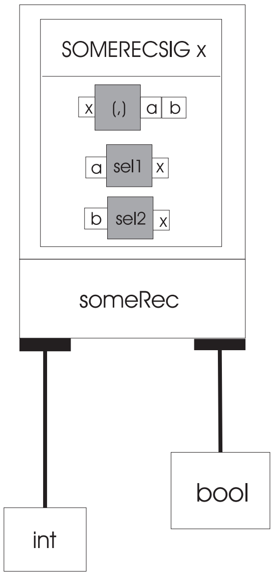

Figure 43: A FAD record type model

FAD Model

A record is presented in FAD in a similar way to a tuple. The constructor signature associated with the type also includes the selection functions. The constructor signature could be created through inheriting a tuple constructor signature, which reinforces the fact that a record is a tuple with some extra functionality. A constructor will be applied to named parameter types, and the signature will be extended with the relevant selector functions. We illustrate this with the FAD representation of `someRec`, the type of the value `aRec`, in Figure 43.

Extensible records can thus be naturally represented through a type associated with a permissive signature, with extensions declared through signature inheritance.

6.3 Algebraic Types

Algebraic or concrete types are either built in to the implementation language, such as the Booleans, or are declared by the user. Each new algebraic type is declared using a type constructor such as the Haskell type constructor `Maybe`. Its values are constructed through one or more value constructors which are declared with the type constructor. Algebraic types were fully described in Section 3.1.5.

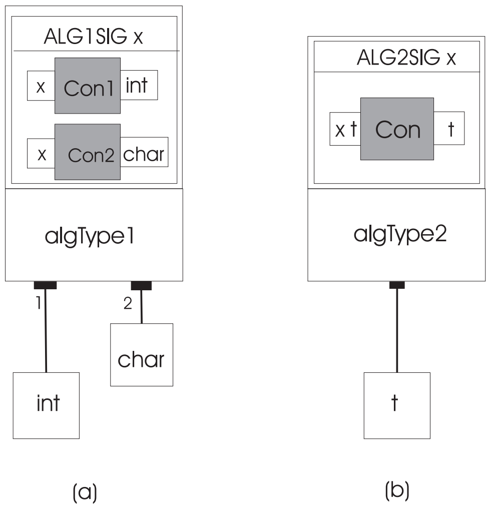

Figure 44: A FAD Algebraic Type

Any algebraic type with at least one non-zero arity value constructor uses at least one type. That is, some values of the type are created by applying one of its value constructors to a value or values of particular types. The sum type `AlgType1` uses values of type `Int` or `Char`, and the parameterised type `AlgType2 t` uses values of any type `t`.

```
data AlgType1   = Con1 Int | Con2 Char
data AlgType2 t = Con t
```

See Section 3.1.5 for a more detailed description of algebraic types.

FAD Model

An algebraic type instantiates a permissive signature that specifies the constructors of the values of the type. FAD represents the types `algType1` and `algType2` as presented in Figures 44(a) and 44(b). The names of value constructors begin with an upper-case letter. A sum type is indicated by annotating the use relationship links as described in Section 5.4.8.

6.4 Abstract Type

An abstract type in contrast to a concrete type hides information regarding the construction of values of the type. An abstract type focuses attention on what one can do with values of the type in ignorance of its implementation details. Abstract types are the functional programmers' mechanism for modular development based on encapsulation and abstraction. They achieve encapsulation through preventing access to their implementation, and abstraction by providing an explicit interface.

Abstract data types are therefore integral to the development of a modular system based on information hiding. The methodology encourages designs built on abstract types as will become clear in the following chapter.

FAD Model

FAD supports type abstraction through its modules and exclusive signatures. Every type is declared in a module. Abstraction is achieved through associating with the module an exclusive signature that specifies the type but not its constructor signature. Hence, within the module the type is concrete but when used via the exclusive signature described above, the type is abstract. That is, an entity declared in the same module has access to the type's implementation. Any entity declared in another module whose use relationship is mediated by an exclusive signature that enforces abstraction, does not have access to the type's implementation.

We illustrate in Figure 45 with a model of the following code. The module implementation has been elided for space reasons.

```
module TreeMod(Tree, treeFun1, treeFun2) where
        data Tree a = Nil | Node a (Tree a) (Tree a)
        treeFun1 :: Tree a -> a
                ...
        treeFun2 :: Tree a -> Int
                ...
```

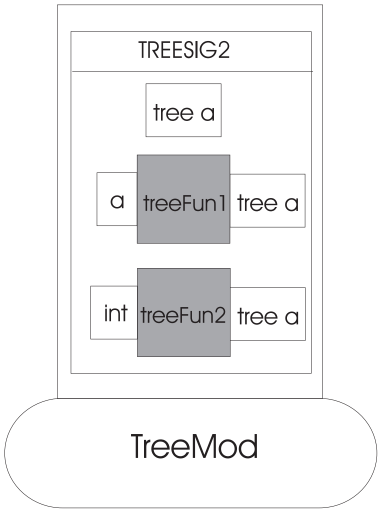

Figure 45: A FAD Abstract Data Type Model

6.5 Polymorphic Functions

Polymorphic functions provide a significant reuse mechanism for functional programmers. *Parametric* polymorphic functions can be applied to values of many types. The type of any polymorphic function includes at least one (unconstrained) type variable of kind $*$, which can be instantiated by any type. That is, a polymorphic function does not require any specific characteristics of the types that instantiate at least one of the type variables of kind $*$ in its type.

FAD Model

One represents a polymorphic function in FAD as a function whose type includes at least one type variable of kind $*$ that is not associated with any permissive signature.

Any function description that does not include any associations with permissive signatures, or only associations with permissive signatures without parameters of kind $*$, could possibly be implemented as a polymorphic function. Full details of the development of polymorphic functions are described in Section 7.3.2. The polymorphic identity function `id` is presented in Figure 46.

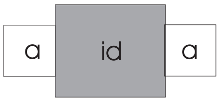

Figure 46: Polymorphic Function Model

6.6 Type Classes, Instantiations and Overloaded Functions

We stated in Section 5.2.3 that a permissive signature associated with a type presents a contract of use for values of that type. That is, the signature is not acting as an interface, in the sense of controlling access to entities of an associated item, but simply as a guarantor that certain functions are defined over the type. That is the *minimum* functionality supported over the type is that declared by the permissive signatures instantiated by the type.

FAD Model

Since type classes (and constructor classes) provide a guarantor service for a set of overloaded functions they are presented as permissive signatures in FAD. Type class instantiation is simply type/permissive signature instantiation in FAD, and class declaration with a non-empty context is supported by permissive signature inheritance. We illustrate both of these situations in Figure 47, in which the following code is graphically represented.

```
class SomeClass a where
        fun1 :: a -> a
instance SomeClass SomeType where
        fun1 = id
class SomeClass a => AnotherClass a where
        fun2 :: a -> a
```

Non-empty contexts can also appear in instance definitions and function definitions. A function with a non-empty context is an *overloaded function*. Each element in the context is represented in FAD as a type/permissive signature contract association. This

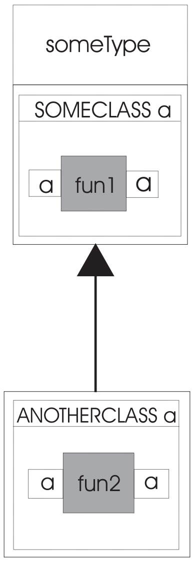

Figure 47: Class Instantiation and Class Declaration

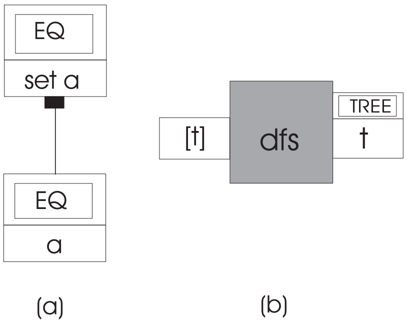

Figure 48: Class Instantiation and Function Definition with Non-Empty Context

is illustrated in Figures 48(a) and 48(b) where the following instance declaration and function declaration are modelled respectively.

```
instance Eq a => Eq (Set a) where...
dfs :: Tree t => t -> [t]
```

6.7 Multi-Parameter Classes

Where single parameter classes are supported by Haskell 98, Gofer and Clean, multi-parameter classes have not been included in Haskell 98, and are only supported by

Gofer (and extensions of Hugs 98 and the Glasgow Haskell compiler). They are however rapidly gathering support in the functional programming community and have been proposed by Peyton Jones [98] for inclusion in the next standard Haskell release. The proposal uses the detailed arguments provided in [102]. We therefore believe that FAD should support multi-parameter classes.

FAD Models

The paper [102] outlines three types of support provided by multi-parameter type classes:

- overloading with coupled parameters
- overloading with constrained parameters and,
- type relations

which we will represent using FAD notation.

Overloading with coupled parameters is the natural generalization of the single parameter overloading supported by type classes. There are many situations where a tuple of types (with each type possibly exhibiting certain behaviours) exhibit a particular set of behaviours, and multi-parameter type classes naturally support such a situation. We present an example from Jones' paper [66], illustrated by the FAD representation in Figure 49.

```
data State s a = ST (s -> (a,s))
class Monad m => StateMonad m s where
        update :: (s -> s) -> m s
instance StateMonad (State s) s where
        update f = ST (\s -> (s, f s))
```

Single parameter type classes in which the parameter is of kind `* -> *` or any non `*` kind, impose no constraints on the type variable(s) associated with any instantiating type constructor. For example, if one wants a set type to instantiate a class which includes a function for combining two items of the instantiating type, then one needs to restrict the set element types to 'equality types' or those that instantiate an

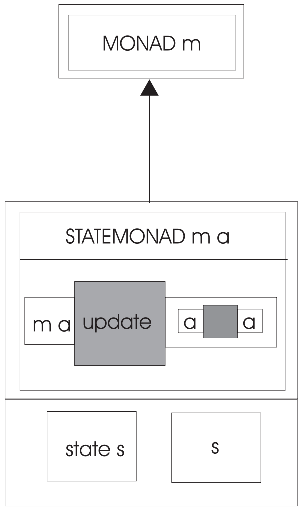

Figure 49: Overloading with Coupled Parameters

equality class. This requires access to the parameter of the type constructor, which is achieved through multi-parameter type classes. This is an example of overloading with constrained parameters.

Overloading with constrained parameters allows the user control over the type variable in a constructor class, in contrast to the single parameter case where the type variable is universally quantified. Hence one is allowed to achieve a higher level of abstraction by creating a type class of generic behaviours, and then support specialization within the context of the instance definition.

Once again we provide implementation code and the corresponding FAD notation in Figure 50.

```
class Multi m a where
        item    :: m a
        combine :: m a -> m a -> m a
instance Class1 a => Multi TypeCon a where
        item    = ...
        combine = ...
```

Type relations allow the user to specify a set of behavioural relationships between two types that are looser than those described in the previous two examples. Liang, Hudak, and Jones [76] present the following example of a class defining an isomorphism

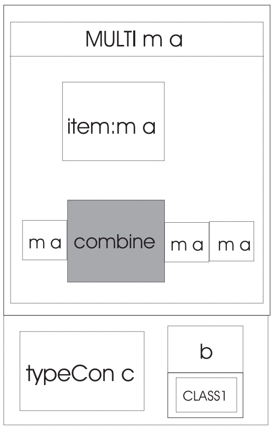

Figure 50: Overloading with Constrained Parameters

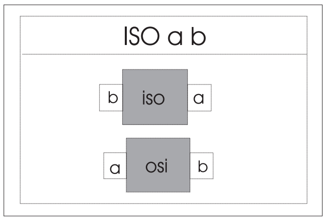

Figure 51: Type Relations

between types.

```
class Iso a b where
        iso :: a -> b
        osi :: b -> a
```

The FAD representation of this class is presented in Figure 51.

6.8 ML Structures, Signatures and Functors

An ML *structure* is a collection of declarations that can include types, functions, values, other structures, and signatures. Each structure can be named and has a default *principal signature* that is the collection of type specifications of the structure's entities. However, one can override this signature through explicitly assigning a declared signature to a structure. That is, ML supports independent modules (structures) and signatures. Thus several new structures can be declared by associating a single structure with different signatures

As with all functional languages, modules are not first class and hence cannot be passed as arguments to functions, returned as results from functions or appear in data structures. However, SML supports *parameterised modules* or *functors* which provide a mechanism for creating new structures from existing ones in an efficient and reusable manner. That is, a functor takes zero or more structures as parameters and returns a structure as a result. Functors with zero arguments are used simply to present a consistent approach to structure development. Where a function is constrained by its type specification, a functor is constrained by the stated signatures of the parameters and returned value.

A structure's signature can be either transparent or opaque, the latter making the type's declared in the structure abstract. Another level of abstraction control is allowed, where the user explicitly declares particular types in the structure abstract. See [88] for full details on SML's modular support.

FAD Model

In FAD we represent a structure as a module and an SML signature as an exclusive signature. An opaque SML signature is represented by an exclusive signature in which any type is specified without its constructor signature. That is, abstraction is represented as described in Section 6.4.

We illustrate these ideas by presenting in Figure 52 the graphical representations of the following ML structures based on those defined in Paulson's *ML for the Working Programmer* [96]. We present the structures and signatures in elided form for space reasons.

```
structure Queue1 =
        struct
        type 'a t = 'a list;
        exception E;
        val empty = [];
        fun enq(q,x) = q @ [x];
```

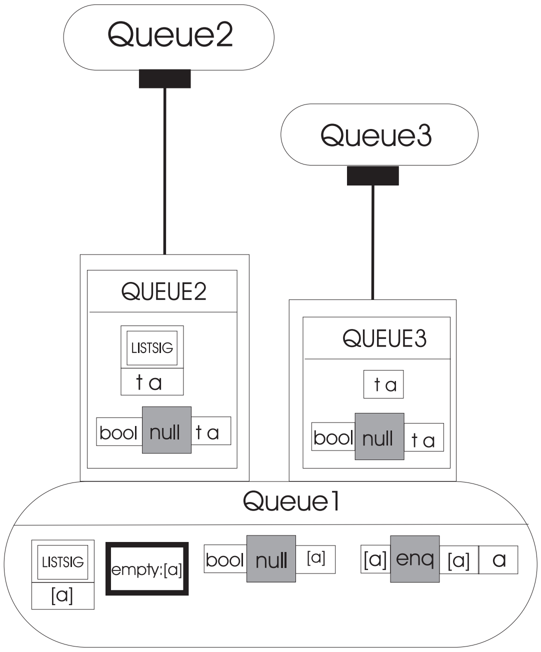

Figure 52: Structures and Signatures

```
          fun null(x::q) = false
            | null _     = true;
          end;
signature QUEUE2 =
          sig
          type 'a t
          exception E
          val null : 'a t -> bool
          end;
structure Queue2 :  QUEUE2 = Queue1;
structure Queue3 :> QUEUE2 = Queue1;
```

When declaring a functor, it is good practice to make explicit the signature that each parameter structure is required to support, and the signature of the returned structure. One cannot model functors directly in FAD but one can model the result of their application. A functor when applied to its argument structure(s), which each support an explicit interface, returns a structure that uses the argument structures and

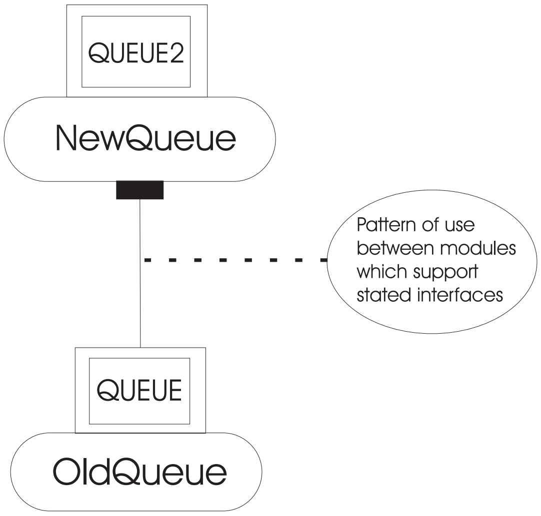

Figure 53: Functor Application Model

itself supports an explicit interface. A functor application can be modelled using FAD's modules, exclusive signatures and the module use relationship.

For example, the structure `NewQueue` is the result of the application of the functor `LimitedQueue` to the existing structure `OldQueue`, and this relationship is represented in FAD as in Figure 53.

```
functor LimitedQueue (Queue: QUEUE) : QUEUE2 =
        struct
          structure Item = Queue;
                  ...
        end;
structure OldQueue : QUEUE
        struct
                  ...
        end;
structure NewQueue = LimitedQueue (OldQueue);
```

One can signal the potential for the implementation of a functor by adding a comment to the diagram that states that the pattern of module development is likely to be repeated.

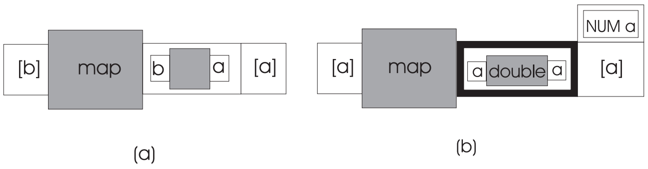

Figure 54: Higher-Order Function Model

6.9 Higher-Order Functions

A function which either takes a function as an argument or returns one as a result, is known as a higher-order function. Thus, by definition all curried functions are higher-order. These are supported in FAD as described in Section 5.4.3.

FAD Model

Functions which take functions as arguments are modelled as functions, with the functional type enclosed in a type rectangle. This is illustrated in Figure 54(a) with the Haskell function `map`. In Figure 54(b) we represent the partial application of `map` to the function `double` which doubles a number. The permissive signature associated with the second argument type, indicates that the function can only be applied to lists of types that support the various arithmetic operators (plus some other functions). The signature only needs to be associated once when there is repeated use of a parameter or type name. Figure 54(a) declares that `map` is defined over all list types and thus can be applied to values of a subset of these types as required by the associated permissive signature in Figure 54(b).

6.10 Existential Types

Existential types (or existentially quantified type variables) are a mechanism for allowing values of differing types in a single data structure. That is, one can create heterogeneous data structures. This is in contrast to *universally quantified polymorphic types* in which each value of the type must itself be monomorphic. That is, one can only construct homogeneous data structures.

However, the use of existential types is restricted. When a constructor with an existentially quantified type is used in pattern matching, the actual type of the quantified variable is not allowed to escape outside the expression tied to the pattern matching. Existential types can therefore only be used in functions where one does not try to access an element of the data structure for external use. For example, a length function that simply takes a list of items and returns the number of items, could be applied to an existentially quantified list type. However, a function that returns the nth element of a list could not be applied to values of an existentially quantified list type, since the actual type of each element is unknown.

Existential types are currently supported by a minority of modern functional languages or implementations of languages. These include Clean and Hugs 98.

Laüfer [74] argues that combining type classes and existential types in a single language delivers significant expressive power. Existential types provide a mechanism for declaring first-class abstract data types, and an associated type class declares the type's interface. We present below an example based on one from [74], which was written using the Chalmers Haskell B. interpreter, HBI [7].

```
data KEY = (KeyClass ?a) => MakeKey ?a
```

Since all type variables that are free and have a name that starts with '?' in a type definition are considered to be existentially quantified, the above declares a data type with an existentially quantified variable that is constrained by the type class `KeyClass`. Thus, the type class `KeyClass` declares the interface to the first-class abstract type `KEY`.

FAD Model

FAD models existential types using types and the type use relationship. One can view an existential type as a non-parameterised type with parameterised value constructors that uses unknown (but possibly constrained) types to construct its values. The FAD representation is presented in Figure 55.

The type `key` uses the values of unknown types signalled by the type variable `a`, which is constrained by the associated permissive signature `KEYCLASS`. It is therefore clear from the model that we have a non-parameterised type using an unknown type in the construction of its values. Thus the type must be an existential type.

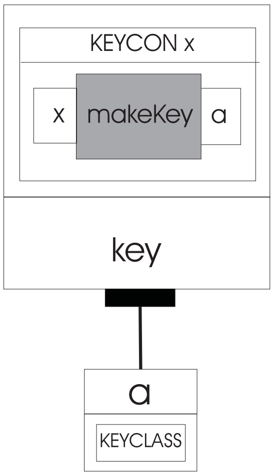

Figure 55: Existential Type Model

6.11 Summary

This chapter has presented a non-exhaustive selection of functional programming designs. We have illustrated how they can be naturally modelled using the modelling language of FAD. In practice it is important for the model of the problem to guide design and not vice versa. In the following chapter we describe the methodology, how it supports the development of an analytical model of a problem, and the iterative development of an implementable design.

Chapter 7

FAD Methodology

In Chapter 5 we presented the units and relationships of the modelling language of FAD. We also provided a syntax and semantics for the models built using these elements. In Chapter 6 we showed how common designs used in functional programming can be naturally modelled in the language. In this chapter we present the techniques of FAD and describe how they fit into an overall methodology. We will use the *football results processing* case study described in Section 5.1, to illustrate elements of the methodology. Each technique will be described by explaining its activities and deliverables. Where appropriate we will clarify how it supports software development within the paradigm as described in Chapter 3, and how it contrasts with object-oriented development as outlined in Chapter 2.

FAD is best used within a process that supports all phases of system development, which are described in detail elsewhere [83, 12]. FAD is a *software* analysis and design methodology and therefore does not deliver any techniques for analysing and designing a system's hardware needs. It provides techniques for analysing the software-specific goals procured through requirements analysis, and techniques for developing a design suitable for implementation in a modern functional language.

FAD can be used in the development of any software that could be implemented in a functional language. That is, its application domain is the same as that for any functional language. This is in contrast to, for example, the Specification and Description Language (SDL) [9, 19], which is best applied to real-time systems, and Jackson's structured programming method (JSP), which is appropriate for serial file processing or information processing, but inappropriate for systems with no dominant information structure [61, 22].

We described in Chapters 5 and 6 how the modelling language supports *inter alia* functions (first-order and higher-order), abstract datatypes, parametric polymorphism, type classes (including single and multi-parameter), SML structures and functors, and modules. In this chapter we will describe how the methodology facilitates the discovery, use and reuse of the building blocks and glue of the functional programming paradigm.

The techniques are described within a methodology since we are not simply presenting a collection of techniques to be applied in an ad-hoc manner. Rather we have specified a modelling language through which models are described, and present guidance on the application of the techniques and how their input requirements and deliverables are related. This will be emphasised in this chapter as the description of each technique will include details of both required inputs and deliverables.

7.1 FAD's Phases and High-Level Process Models

The methodology is divided into two main phases, analysis and design. However, this neither implies a strict division between the two phases, nor a linear application of the techniques within the phases. We believe that FAD is best applied within an iterative and incremental development approach. Thus, for example, one could develop on the basis of a subset of functional requirements and then iteratively develop as additional requirements are introduced. Since FAD will use the same models, notation and diagrams to support all parts of development through analysis and design, the developer is free to decide on the chronology of the application of the methodology's techniques. A methodology with phase-linked models penalises the user for backtracking, since later models that require significant effort in construction will require reconstruction. When one has models and notation that are applicable throughout development, although any change still requires work on the part of the developer, this work tends to focus on the modification of existing models and other supporting documentation.

Most structured methods have phase-linked models and have historically been used within a waterfall development process, which was first described by Royce [118]. This process is inherently linear in nature and has been criticised for:

- not adequately addressing changes;
- assuming a relatively uniform and orderly sequence of development steps; and,
- not providing for such methods as rapid prototyping. [58]

These shortcomings have been addressed both by Boehm's Spiral Model [13], which explicitly addresses the use of prototyping and other risk-resolution techniques, and the iterative and incremental process typically encouraged when using OOADMs. Here one separates the system into subsystems that can be delivered incrementally, and encourage an iterative approach to the development of a system's entities. Prototyping is also encouraged within an iterative approach to software development. The debate here tends to focus on the choice between *same-language prototyping* and *different-language prototyping* [114].

The reason for using the classification into the two phases of analysis and design, aside from simplifying exposition, is twofold. Firstly, although the methodology should not be applied in a strictly linear fashion, there is a general linear movement through the methodology which is highlighted by making these subdivisions. That is, initial techniques are largely analytical in nature with design issues gradually taking precedence as development proceeds. Secondly, some of the techniques, such as *scenario analysis*, span more than one phase and cannot be optimally described without reference to their use in each phase. Scenario analysis, to be described in Section 7.2.2, is a technique of FAD that is initially used to investigate the major uses of the system, but will later be used in the design of functions. That is, some techniques have phase-linked rôles.

The application of FAD is linear in another sense. The early stages of analysis will take non paradigm-specific requirements and describe them using the paradigm-specific constructs, functions and types. As the system is developed, the ties to the paradigm will become stronger, resulting in a model which is best implemented in a functional language. When the implementation language is known, one can (iteratively) develop designs that reflect the characteristics of the implementation language. This is clearly a sensible approach, given that the early analysis part of any methodology needs to model the problem free of any implementation language constraints, whereas the latter design stages should be seeking an efficient, effective and maintainable solution. All these issues should become clearer as the methodology is described.

Phase	Task	Techniques
Analysis	Describe major uses as a collection of functions.	Functional Requirements Analysis
	Investigate each 'use function' and describe type and function dependencies, and new 'use functions'.	Scenario Analysis Type Dependency Analysis
	Develop initial subsystem architecture and assign types and functions to subsystems.	Subsystem Architecture Analysis Type/Function Host Analysis
	Further analyse functions/types with inter-unit relationships.	Scenario Analysis Type Dependency Analysis
	Develop exclusive signatures.	Exclusive Signature Analysis
	Develop initial prototype.	
	Investigate subsystem 'use functions'.	Scenario Analysis Type Dependency Analysis
	Develop module architecture for each subsystem and assign types and functions. Develop exclusive signatures.	Module Architecture Analysis Type/Function Host Analysis Exclusive Signature Analysis

Table 1: FAD Methodology – Analysis Phase

We will therefore present the methodology within two main sections titled *Analysis* and *Design*. In describing each technique, we will present the possible documentary deliverables, leaving it to the developer to decide what is actually appropriate for a given project.

The methodology will be described as a collection of tasks within each phase using a linear presentational style. Each task is executed either through a single technique or several techniques. Since several of the techniques span more than a single task, each new technique will be defined where it is first introduced. However, we also describe the application of each technique as it is used. FAD's analysis phase is summarized in Table 1, where we present the tasks of the phase and the techniques used to execute each task.

7.2 Analysis

Analysis focuses on modelling system requirements using the units and relationships of the modelling language. One should be focusing on *what* is required rather than *how* it will be delivered. However, in any paradigm-related ADM one is unable to totally separate the *what* from the *how*. For example, object-oriented methodologies describe user requirements in terms of the objects which host the methods whose collaboration implements each requirement. Function or action-oriented structured methodologies describe user requirements through data flow diagrams and thus in terms of independent data and processes [37, 152]. Data-oriented structured methods model user requirements through their effects on the data of the system [73, 23]. In all cases, one is forced into making paradigm-related design decisions.

FAD supports software development within the functional programming paradigm and thus user requirements will be described in terms of functions where data flow is made explicit. The initial emphasis during analysis is on the modelling of user requirements. Issues of implementation efficiency, reusability and maintainability are of increasing importance as development proceeds.

FAD, in common with several use-case dependent OO methodologies [63, 64], is a *user-driven* methodology in that users' functional requirements dominate initial development. Users could be humans, hardware devices or another system. Initial techniques clarify the major uses that the system needs to support, and then investigate each in turn. FAD encourages an iterative approach to development. One therefore may focus initially on a subset of the major user requirements, develop the system to satisfy these requirements, and then return to add extra functionality to the system. The technique that analyses the system's requirements and returns a list of the users' functional requirements is *functional requirements analysis*.

7.2.1 Functional Requirements Analysis

This technique takes as input the system's requirements and returns the major functional requirements of the system users. These are modelled as functions. A detailed discussion of *requirements engineering* is beyond the scope of this thesis but is comprehensively described elsewhere [129]. Each function is declared in a FDD with its

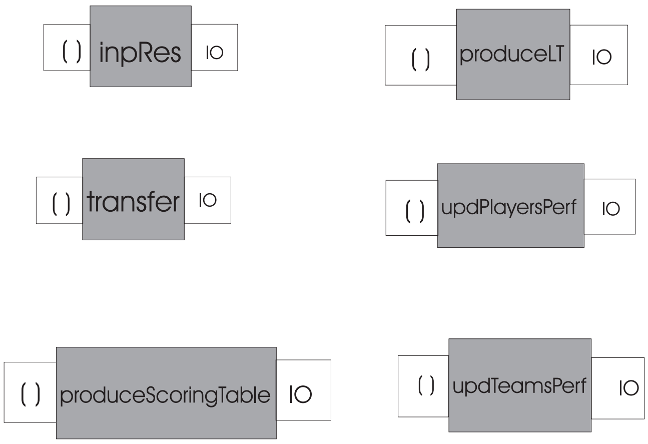

Figure 56: User Requirements Functions

argument and results types recorded. This immediately emphasizes the explicit nature of data flow within the paradigm. Figure 56 presents the functions that describe the user's - in this case a data entry clerk - functional requirements for the *football results processing* system.

To simplify exposition, each function is specified as a text-based I/O function. We are not, however, enforcing a particular user interface on the system. The modular approach to development, encouraged by FAD and supported by modern functional languages, will enable an alternative (possibly GUI) interface to be introduced if required. The important issue here is clarifying the user's requirements.

Pure functional languages have developed various mechanisms for dealing with the impurity of I/O such as continuation passing, stream processing and most recently monadic I/O [53, 103]. The monadic approach is popular since it presents a pattern of computation that is not restricted to I/O alone and because

> [By using monads] we have the intuitive sequential nature of imperative input/output and the uncluttered code style that results from using global variables, but have neither the referential opacity conveyed by both these things in an imperative language, nor the excessive heavy framework and lack of expressive expression forms which such languages have.[52]

However, since FAD is not tied to a specific implementation language, one is free to describe I/O functions in one's own terms, as long as it is supported by clear, unambiguous documentation. We have chosen here a notation that is similar to that used in the monadic I/O of Haskell [103] but is not meant to signal any particular approach to I/O implementation. An I/O function has an argument type named `IO` (written in upper case to indicate that this is not a typical type) and a return type that depends on the function's characteristics. For the above functions, the return type is `()`, which is the type with a single value of the same name. This type specifies a function that does some I/O and returns the value `()`. I/O functions that return a value of some other type, such as a string, are similarly specified with a return type `string`. Of course, if one wants to develop a system in which I/O is delivered monadically one can make this explicit by associating the permissive signature `MONAD` with the `IO` type. This however is a design decision which is typically applied later in development, possibly when one is tailoring a design to a particular implementation language. We describe the development of permissive signatures in Section 7.3.1.

The six functions that describe the user's functional requirements are:

- `inpRes` which implements the result input functionality;
- `produceLT` which manages the production of a league table;
- `transfer` which implements the transfer of a player between two football clubs;
- `produceScoringTable` which implements the production of a scoring table;
- `updPlayersPerf` which updates a player's performance data given recent matches; and,
- `updTeamsPerf` which updates the performance data of teams involved in recent matches.

Each function will be documented in a *function description document* (FDD). Initially there will be little documentation beyond the function's name, argument and result types. However, the FDD is an appropriate host for a textual description of the function's purpose. This is illustrated in Figure 57 with the initial FDD for the I/O function for producing a league table, `produceLT`.

Interested parties are informed of the initial collection of 'major use functions' in order to confirm that the collection is complete and correct. Upon confirmation, a

Function Description Document	Football
Name:	`produceLT`
Version:	19990620:0
Module:	
Arity:	1
Type Specification:	`IO -> ()`
Contract Association:	
Instantiations:	
Functions Used:	
Description:	

The user requests the production of the current league table. The table is generated from the existing team data that is stored on file. Each football team hosts information regarding its performances, which is selected and used to generate a league table entry. This entry includes the points achieved by the team. The complete league table is created from the league table entries for each team where the position in the table is first determined by the number of points, followed by goal difference, goals scored, and finally alphabetically. Each league table is stored in a file with previous league tables.

Figure 57: Initial Function Description Document for `produceLT`

decision needs to be made regarding how one proceeds. One can either adopt a 'big bang' approach and investigate all of the functions, or focus on a subset and return to others later during development. The 'big bang' approach is appropriate if one is dealing with a system with relatively few user requirements. However, if there are a significant number then one should adopt an iterative approach to development.

Two techniques are used to analyse the functions: *scenario analysis*, which investigates functions and, *type dependency analysis*, which investigates types. They are practised in parallel since each function is specified in terms of its type.

7.2.2 Scenario Analysis

Functional programs are built from functions. Thus any model of a system's functionality must be built using functions. Scenario analysis, a technique which is practised at various stages of development, investigates a system's functions and describes them in terms of other functions. Initially one uses the technique to model the major user requirements of the system.

Scenario analysis investigates the behavioural characteristics of a function and describes them in a set of models that are graphically presented in *function dependency diagrams*. Each diagram describes a function in terms of one or more functions to which it is linked via a *function use relationship*. A single function will be described through several function dependency diagrams if the function has conditional behaviour. The functional programming paradigm provides substantial support for function development and reuse and encourages the development of simple functions that are then used to develop more complex functions.

When applying scenario analysis, one should adopt a modular approach where each behavioural requirement of an analysed function is delivered by functions upon which it depends. The dependency is not an implementation dependency but a behavioural dependency. That is, a function *depends* on the behaviour implemented by the functions it uses. By adopting a modular approach, any implementation changes remain local and thus small scale. This increases the potential for reuse of existing functions, which is supported by FAD as described in Chapter 8.

The approach here is similar to that of *use case analysis* as introduced by Jacobson in his Objectory method [64]. Although use case analysis is a popular component of various OOADMs - it has recently been adopted for use within the Unified Software Development Process using UML as the modelling language [63, 127] - its prime focus is modelling user interactions with a system which are, of course, functional in nature. Thus one can argue that it sits more naturally within a functional development methodology. Using an OO methodology one is required to deliver the results of use case analysis in a manner consistent with the paradigm. Thus every function or method is required to be the responsibility of a class, which forces early decisions regarding the assigning of methods to classes. We will not present a description of use case analysis here but instead will describe *scenario analysis* and support its description with examples from the development of the football system. Use case analysis is described in Section 2.3.

Scenario analysis takes as input the description of a particular user requirement such as that presented in an initial FDD. However, further information may be required, which could be delivered verbally, graphically or in some textual representation such as informal English, pseudocode or a formal language. Each analysis returns one or more dependency diagrams and accompanying supporting documentation in the form of description documents for the entities in the diagram(s).

To illustrate scenario analysis we present an analysis of the function `produceLT`, which is informally described as below.

> The user requests the production of the current league table. The table is generated from the existing team data that is stored on file. Each football team hosts information regarding its performances, which is selected and used to generate a league table entry. This entry includes the points achieved by the team. The complete league table is created from the league table entries for each team where the position in the table is first determined by the number of points, followed by goal difference, goals scored and finally alphabetically. The latest league table is then appended to the file which hosts the previous league tables.

One possible model of the function `produceLT` uses three functions: `readTeamsFile`, which retrieves the latest team data from a file; `generateLT`, which takes the collection of teams and returns a league table; and, `appendLTToFile`, which appends the latest

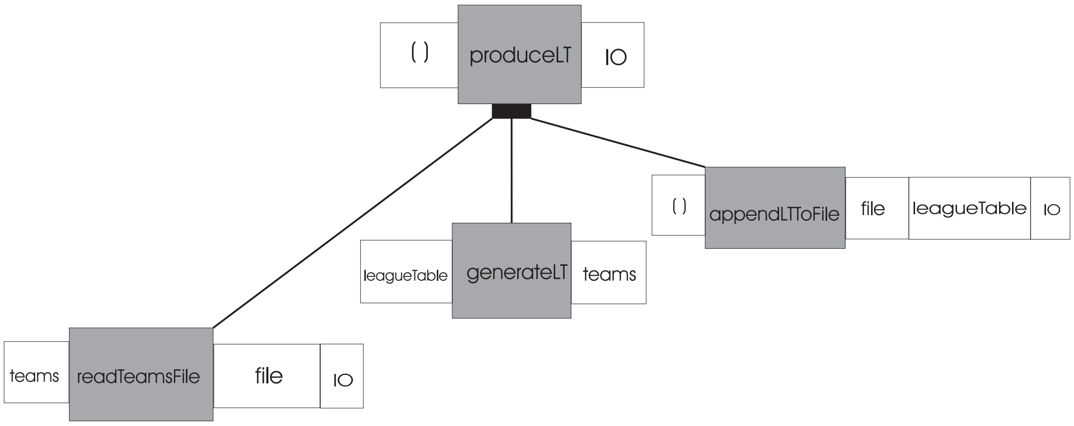

Figure 58: Initial Function Dependency Diagram for `produceLT`

league table to the file that records the history of league tables. Each function is specified in terms of its type and it is incumbent on the developer to clarify the description of each type that is used. The type `file` used by `readTeamsFile` and `appendLTToFile` could simply be a type of filepaths or could be a record like the Haskell library type `Handle`, which includes properties that state whether a file can accept input and/or output, or whether buffering is enabled or disabled and in what form [101].

As indicated previously each type used by a function will be investigated using type dependency analysis, which we describe in Section 7.2.3. Each type dependency analysis delivers a model that is represented in a *type dependency diagram*, a graphical representation of a type and its dependencies.

The initial function dependency diagram for `produceLT` is presented in Figure 58. The function `generateLT` takes an argument of type `teams` for which a *type dependency diagram* is presented in Figure 59. It is clear from the type dependency diagram (and associated documentation) that the type `teams` provides the required input for the function. The FDD for the function `produceLT` will be updated as a result of the scenario analysis, and FDDs and TDDs will be initiated for the new functions and types.

The second illustrative analysis is applied to a function that exhibits conditional behaviour. `inpRes` is the I/O function that supports the user's requirement to input a new football result. An informal description of the function's behavioural requirements is presented below.

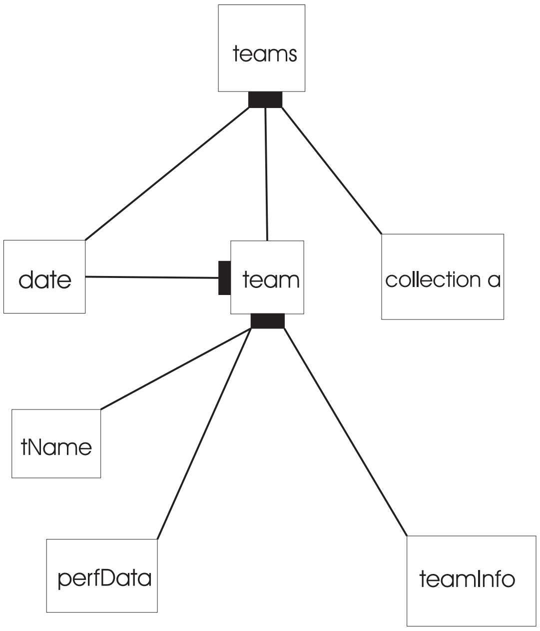

Figure 59: Initial Type Dependency Diagram for the Type `teams`

> Upon initiation by the user, a result is read in as a string that is then parsed. If the parse is successful the parsed result is converted into a result value. If the parse fails, then the user is informed of this failure and the interaction is terminated. The current collection of results are read from file as is the current collection of teams. The (successfully parsed) result is then checked for the existence of the teams and non-existence of the result, and if OK one proceeds by reading the current collection of results from file. If the result fails the check the user is requested to edit the result, which then initiates the process again. An OK result is added to the current collection of results that are then written to the results file.

This scenario is modelled in three function dependency diagrams presented in Figures 60, 61, and 62. Figure 60 presents the dependencies where both the parse and the result check were successful. That is, the result is inputted as a `string` using `readInp`. The `string` is parsed using `parseRes` which returns a successful parse of type `parsedRes`. A `result` is created using `createRes`. The results history and current teams data are retrieved from file and the inputted result is tested for acceptability by `resultCheck`. A successful check is followed by the inputted `result` being added to the current collection

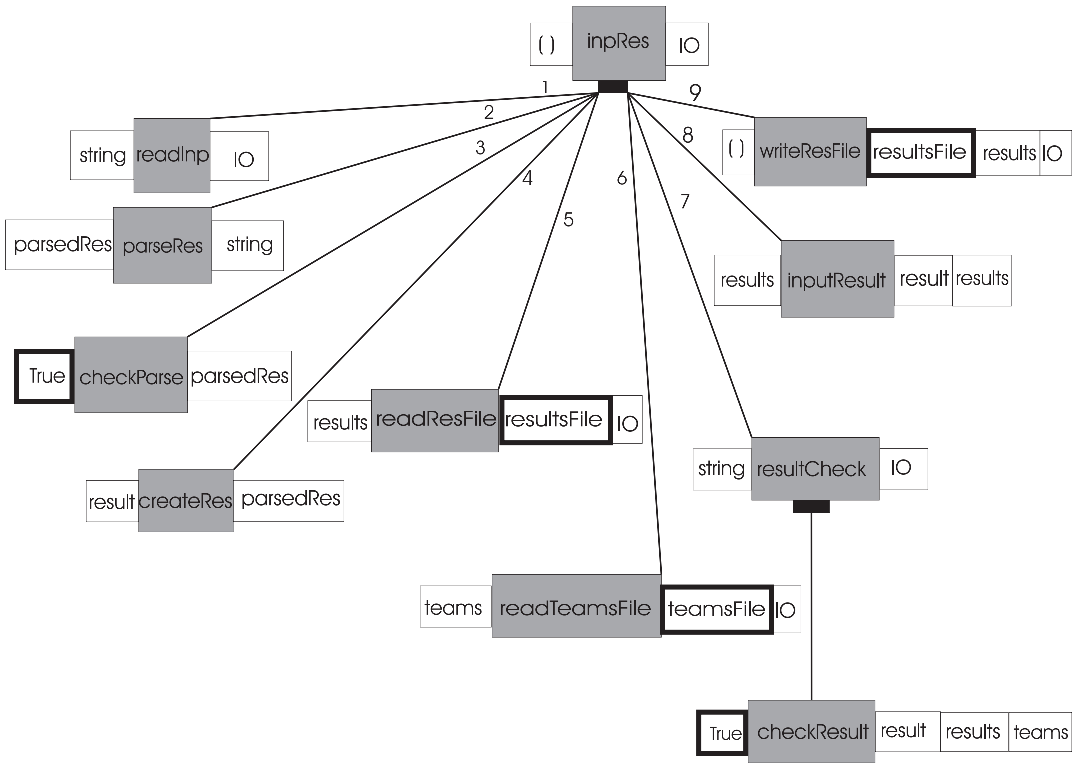

Figure 60: Dependency Diagram for the Successful Case of `inpRes`

using `inputResult` and the new collection of `results` is returned and written to file using `writeResFile`.

In Figure 61 we present the dependency diagram which represents the case when the parse fails and results in the function `failedResParse` being called. A failed `resultCheck` where there is an error in the inputted result is described in Figure 62. In this diagram we have left out the functions preceding the result check since these are represented in Figure 60. We have also used a comment to indicate a looping design.

The functions that model a scenario analysis are dependent on the types that they use. It is therefore important that these types are analysed in parallel using the technique *type dependency analysis*, which is described in the following section.

7.2.3 Type Dependency Analysis

A type dependency analysis takes a type description and returns a model of the type being analysed. A type is described in terms of the types it uses in the construction of

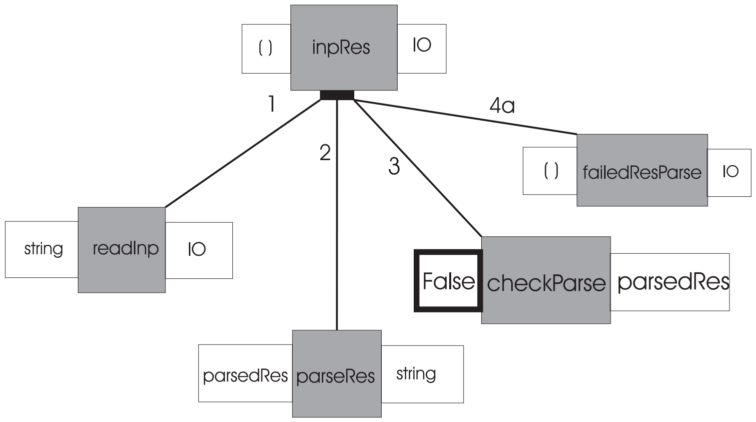

Figure 61: Dependency Diagram for the Failed Parse `inpRes`

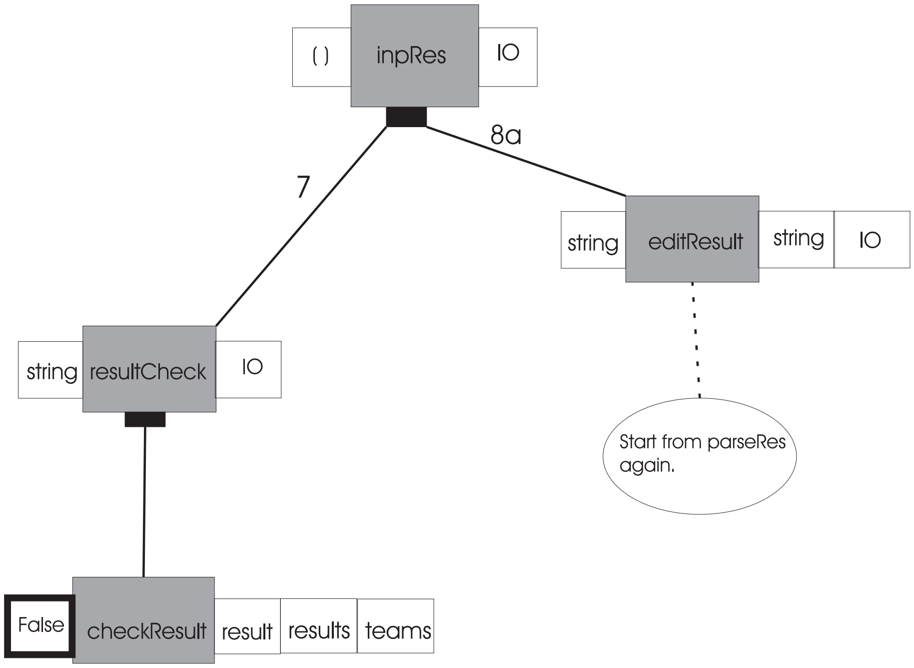

Figure 62: Dependency Diagram for the Failed Result Check Case of `inpRes`

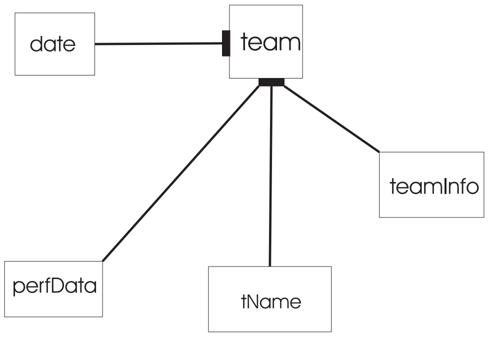

Figure 63: Type Dependency Diagram for the Type `team`

its values. As development proceeds, the model may also include details that reflect the system's modular architecture and behavioural requirements of the types signalled by associated permissive signatures.

We illustrate the technique with the analysis of the type `team`. An informal description is presented below and its type dependency diagram is presented in Figure 63.

> A value of the type team represents a football team. Each team has a unique name and a record of the team's season's performances. In addition, standard team details such as the manager and average home attendances are recorded. Each team value also has an associated date that records the last date of data entry (assuming at most one entry per day).

Thus the type can be constructed using four other types `tName`, `date`, `perfData` and `teamInfo`, which represent football team names, dates, team's performance data and the non-performance data of football teams. In common with function development, where possible a type should be built from existing types reflecting the significant type development support afforded the functional programmer. This approach maximizes the potential for reuse of existing types whose storage and discovery we describe in Chapter 8. The information presented in Figure 63 is recorded in the TDD of Figure 64.

Type dependency analysis, in common with scenario analysis, spans more than one phase and one task of FAD. Initially it is used to describe the types used by the functions returned by scenario analysis in order to confirm that all the required information is

Type Description Document	Football
Name:	`team`
Version:	19990619:0
Kind:	*
Module:	
Types Used:	`date`, `tName`, `perfData`, `teamInfo`
Parameters:	
Permissive sigs.:	
Description:	

A value of the type team represents a football team. Each team has a unique name and a record of the team's season's performances. In addition, standard team details such as the manager and average home attendances are recorded. Each team value also has an associated date that records the last date of data entry (assuming at most one entry per day).

Figure 64: Type Description Document for the Type `team`

supplied by values of the type. Later it will be used within the design phase as input into the design and implementation of types.

There is a similarity between the use in various structured methods of data flow diagrams (DFDs) and entity-relationship diagrams (ERDs) [26] or logical data structure diagrams [41], and the use here of function dependency and type dependency diagrams. Whereas data flow diagrams focus on the manipulation of data by various processes, ERDs, or data structure diagrams, describe details not supported by DFDs such as the structure of major data entities and their interdependencies [37]. They tend to be used in systems that are reliant on major data or file structures such as database systems. Functional programming's reliance on functions with no side effects and therefore explicit data flow, requires that significant attention is always paid to the types of the functions that deliver the required functionality of the system.

The collection of models generated through scenario analysis and type dependency analysis provide inputs for *subsystem architecture analysis*, which delivers a subsystem architecture for the system. If one is building the system initially on the basis of a subset of the users' functional requirements then one is building an architecture that will need to support future iterations of development.

7.2.4 Subsystem Architecture Analysis

Scenario analyses and type dependency analyses could be applied *ad infinitum* or at least until every function is described in terms of a collection of simple, atomic functions and every type described similarly. In a large project this process can soon become unwieldy and thus one needs guidance regarding termination of the process. A division of the system into manageable units that can be developed independently provides both a structure for future development and guidance regarding the termination of the initial set of scenario and type dependency analyses.

Subsystem architecture analysis takes the deliverables of the previously applied analyses, and returns a project partitioned into several subsystems. The partitioning criterion is *information hiding* [95] through *encapsulation* and *abstraction*. That is, each subsystem hides the details of its design from its clients, who simply require knowledge of the entities available for use. One can therefore develop systems incrementally and use the components beyond the immediate application for which they are being developed.

Such a system will have cohesive units that are loosely coupled. That is, by grouping related abstractions within a subsystem (or module), and by minimising the dependencies between them, one builds a system through independent and focused components.

In addition, information hiding is invaluable as a development tool since it applies the *principle of least commitment* to program design [1]. That is, one can delay design decisions in the knowledge that it neither delays nor harms the development process. Each subsystem's development will be assigned to a development team. The information required of any other subsystem is presented in an associated *exclusive signature* that acts both as a mediator of usage and a specification for development.

We will illustrate this technique with an analysis of the case study. The project can be partitioned into five subsystems that deliver:

- the interaction with the user, `UISS`;
- the parsing functionality required to deal with the various entered data, `ParseSS`;
- the file handling requirements which have been alluded to in the description of various functions, `FileSS`;
- the football-related functionality, which is unique to the case study problem, `FootballSS`; and,
- some general entities which are either typically supported by the standard environment of an implementation language or need to be accessible to all entities of the system, `GeneralSS`.

Each of the subsystems are likely to support functionality that is non-problem specific. For example, `ParseSS` is a subsystem that supports parsing functionality. Functions of the subsystem will support the parsing of values of various types (not just the `string` type) and for a range of grammars. Any required functions will be specified in an associated exclusive signature that hides implementation details. That is, the implementation of the parsing functions (possibly via parser combinators or even monadic parser combinators) is left to the code writers and is likely to be dependent on the implementation language. This model is graphically represented in the *subsystem dependency*

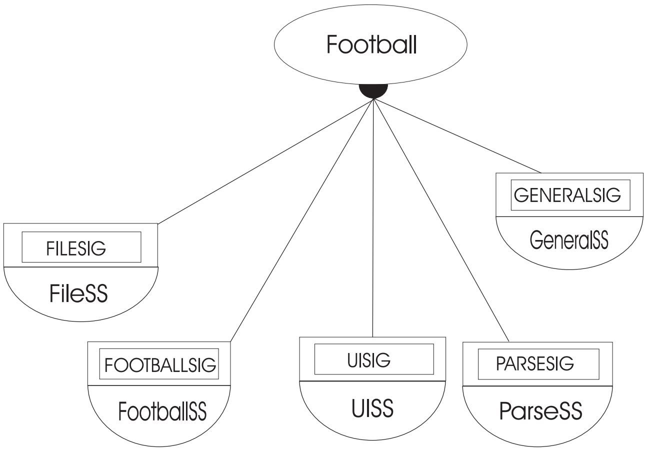

Figure 65: Subsystem Diagram for the Project `Football`

diagram of Figure 65. Each subsystem is associated with an exclusive signature that are currently vacuous.

FAD does not provide a standard blueprint for architectural design as does, for example, Coad and Yourdon's method [29], or metrics for comparing one design against another. However, it encourages modularity through information hiding, which if practised, will result in sensible, reusable designs. Thus if a function is declared in subsystem **S** and its argument and result types are declared in subsystem **T** this suggests poor design with a high degree of coupling between the two subsystems. The models developed through the application of FAD will provide an early indication of (potentially) poor designs.

Type/function host analysis takes the current sets of types and functions and assigns each one to a subsystem of the project. One can then analyse their various dependencies that will be described either as an *intra-subsystem dependency* or an *inter-subsystem dependency*. One wants a design where the former is more frequently in evidence than the later.

7.2.5 Type/Function Host Analysis

Type/function host analysis takes the types and functions described through earlier analyses and assigns each to one of the subsystems. That is, each entity is the responsibility of the development unit that develops the host subsystem. Type/function host analysis is also applied later in development when a subsystem's entities are assigned to modules of the subsystem. The analysis returns updated function and type models whose use relationships reflect the assignment of entities to subsystems. With OO development data and the methods that act on the data are the responsibility of a single object. Through this mechanism one achieves data protection and localisation of control. In functional programming the motivation for assigning entities to modules or subsystems is to manage development and to support the reusability of components of a system. Modules and subsystems host a collection of entities but do not provide a single unit which can be the argument of a function or returned by a function.

Each subsystem will be documented in a series of *subsystem description documents*. A record of the assignment will be written in new versions of the description documents of the assigned entities. Every micro unit will eventually be assigned to a module of the subsystem and the description documents will be updated to reflect this assignment. After presenting an illustrative example from the case study, we describe how the deliverables of this analysis signal where it is necessary to apply further scenario and type dependency analyses in advance of the development of each subsystem.

We illustrate this technique with the analysis of the type dependency diagrams for the function `inpRes` presented in Figures 60, 61 and 62. The results are presented in Tables 2 and 3 where each entity is presented with its host subsystem and some brief commentary.

The information in Tables 2 and 3 is captured in updated dependency diagrams. The use relationships now reflect whether the related units are of the same subsystem or of different subsystems. Inter-subsystem use relationships between two functions or two types are represented by a broken line link. See Sections 5.4.1 and 5.4.8 for a description of the various use relationships. These updated diagrams (presented in Figures 66, 67 and 68) give a clear view of the impact of modular decomposition on the system.

Each subsystem will be developed to satisfy the requirements specified in any associated exclusive signatures, and in the knowledge that other subsystems will provide the types and functions specified in the exclusive signatures through which they are used. It is therefore essential that inter-subsystem dependencies are made explicit at this stage. These will provide input into the development of exclusive signatures that we describe in Section 7.2.6.

Function	Subsystem	Comment
`inpRes`	`UISS`	An I/O function.
`readInp`	`UISS`	An I/O function.
`parseRes`	`ParseSS`	A parsing function.
`checkParse`	`ParseSS`	A function that checks whether a parse is successful.
`createRes`	`FootballSS`	A function that constructs a value of type `result`.
`resultCheck`	`UISS`	I/O function.
`checkResult`	`FootballSS`	A function that tests a value of type `result`.
`readResFile`	`FileSS`	File-handling function. Uses and requires 'readability' of type `results`.
`inputResult`	`FootballSS`	A function that implements a behaviour over the type `results`.
`writeResFile`	`FileSS`	File-handling function. Uses and requires 'writability' of type `results`.
`failedResParse`	`UISS`	An I/O function.
`editResult`	`UISS`	An I/O function.

Table 2: Function Host Analysis for the Function `inpRes`

Type	Subsystem	Comment
`IO`	`UISS`	I/O type.
`()`	`GeneralSS`	Basic type.
`string`	`GeneralSS`	Basic type.
`parsedRes`	`ParseSS`	Parsing type.
`result`	`FootballSS`	Football type.
`bool`	`GeneralSS`	Basic type.
`file`	`FileSS`	File-handling type.
`results`	`FootballSS`	Football type.

Table 3: Type Host Analysis for the Function `inpRes`

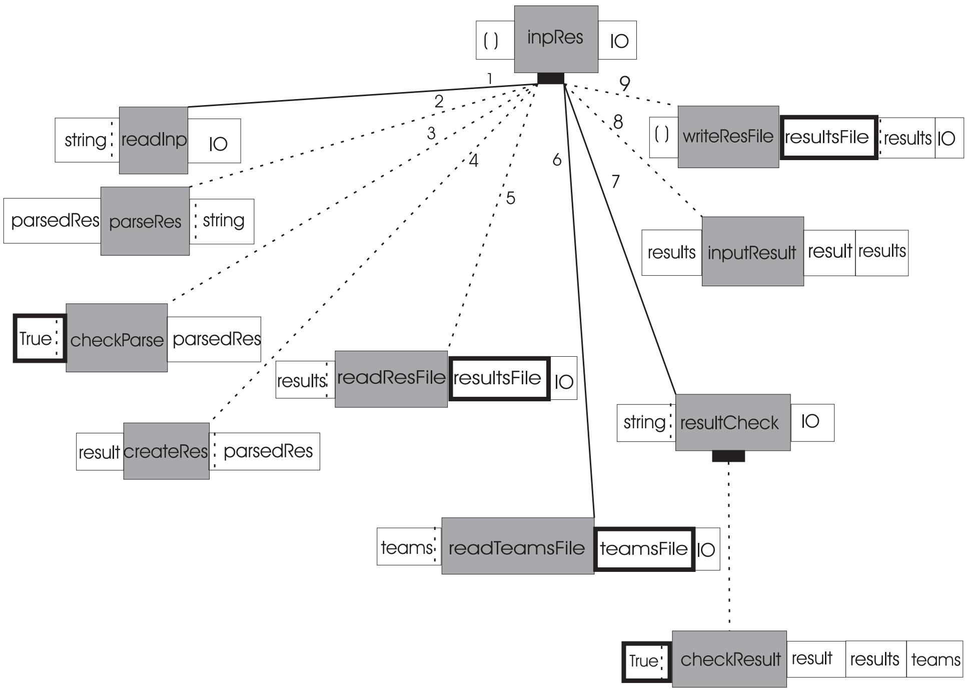

Figure 66: Updated Successful Dependency Diagram for `inpRes`

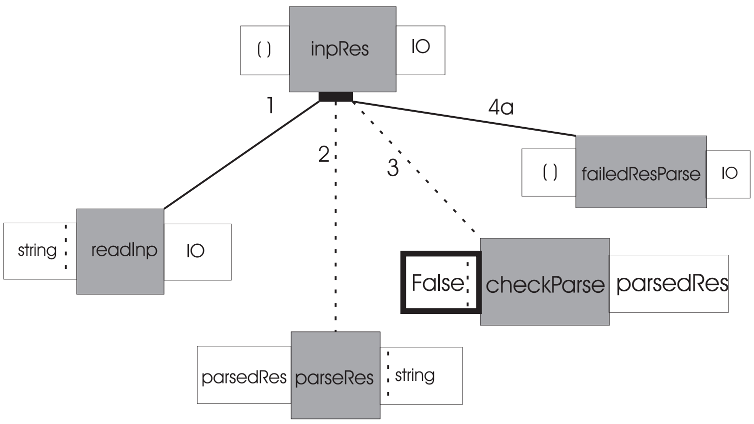

Figure 67: Updated Failed Parse Dependency Diagram for `inpRes`

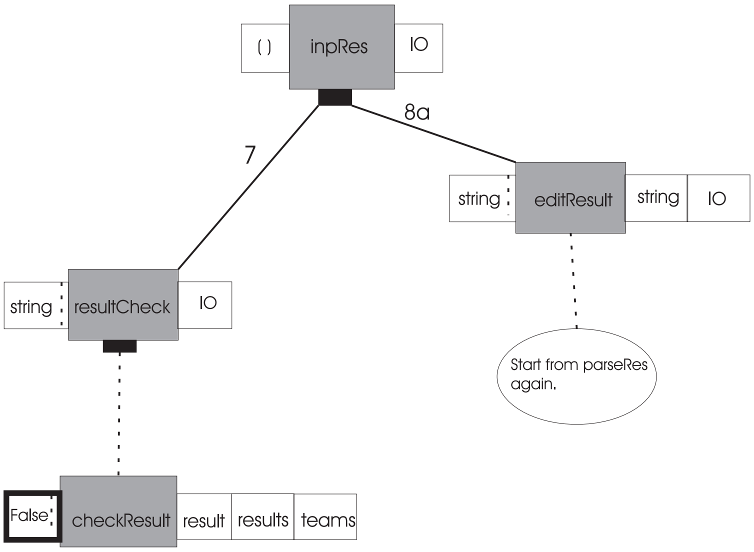

Figure 68: Updated Failed Result Dependency Diagram for `inpRes`

Any inter-subsystem, function/type use relationship indicates that the function should be subjected to further analysis since the function may use other functions which exist in the used subsystem. For example, if the type is abstract various 'get' and 'set' functions may need to be provided. Any other functions should be briefly analysed to confirm that all used functions and types will exist in the same subsystem or in the universally accessible subsystem `GeneralSS`.

We illustrate with some examples from the case study. The function `createRes`, which takes the parsed result of type `parsedRes` and returns a value of type `result`, is assigned to the subsystem `FootballSS`. The function uses the type `parsedRes` of the subsystem `ParseSS`. Assuming that the type `parsedRes` is abstract relative to the function, it will need to be accompanied by functions that return the team name, goals scored and other information required to construct a value of type `result`.

The second example is an analysis of the related `FileSS` functions `readResFile` and `writeResFile`. They both use the type `results` from the subsystem `FootballSS`. Their respective behaviours include the conversion from (and respectively to) a printable `string` representation of a value of type `results`, to (and respectively from) its actual value. They therefore depend on functions that implement this behaviour, which we call `readResults` and `writeResults`. Both functions are assigned to the

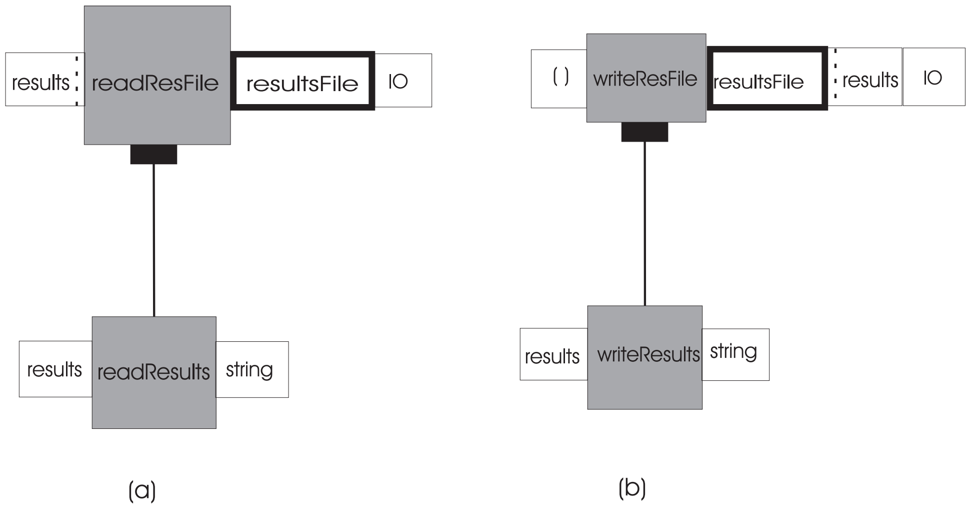

Figure 69: Read and Write Dependencies

subsystem `FootballSS` since they implement behavioural requirements of the type `results`. Alternatively we could declare that the type `results` must instantiate respectively the permissive signatures `READ` and `SHOW` in the declarations of `readResFile` and `writeResFile`. `READ` includes a specification of a simple read function and `SHOW` provides a simple write function. Either approach describes the same model. The analysis and use of permissive signatures is described in Section 7.3.1 within the design phase of the methodology. The dependencies described above are presented in Figures 69(a) and 69(b).

The functionality delivered by a subsystem and required of other subsystems by clients is currently hidden within function and type models that are best used to express particular functionality and type structure respectively. For an accurate view of subsystem functionality and interaction one needs to add interface details to the subsystem model. This is achieved through *subsystem exclusive signature analysis* the results of which are represented in updated subsystem dependency diagrams.

7.2.6 Subsystem Exclusive Signature Analysis

ADMs provide mechanisms for the division of a system into manageable components that can be developed independently. They also provide the mechanisms for gluing the components together to deliver a single system. The glue provided by FAD are subsystem (and module) use relationships. A use relationship links a client subsystem to another subsystem that provides services that are specified in an associated exclusive signature. That is, the interaction between entities of different subsystems is marshalled through a collection of interfaces emphasising the information hiding approach to modular development.

Subsystem exclusive signature analysis takes the various function and type models and filters out those entities that are used via an inter-subsystem relationship. These entities should be specified in the exclusive signature that is associated with their subsystem and mediates access to entities of the client subsystem. Thus exclusive signature analysis returns exclusive signatures that provide a specification for the development of their associated subsystem. They also make explicit the entities of subsystems that are accessible to clients. Exclusive signature analysis returns a subsystem model that includes interface details. If one is looking to build a *prototype* of a system this model provides much of the necessary information.

At this stage we require enough information about each subsystem in order to proceed with the independent development of the subsystems. A single exclusive signature will provide the necessary information even though it will not truly reflect the dependencies between various subsystems. Signatures that provide the interface information for a specific client, *client-specific signatures* will be designed later in development when an accurate description of the system design is required. With an iterative approach to development exclusive signatures are likely to be updated to reflect the addition of new user requirements to the system. We describe the technique that returns client-specific signatures in Section 7.3.5. The updated subsystem architecture for the case study is presented in the subsystem dependency diagram of Figure 70.

The signature `FOOTBALLSIG` mediates the use of entities declared in the subsystem `FootballSS` by entities of the subsystem `UISS`. Included in the signature are specifications of:

- `createRes` and `inputResult`, which are used by the function `inpRes`; and,
- `checkResult`, which is used by `resultCheck`.

In addition, the subsystem `FileSS` uses the subsystem `FootballSS` through the same

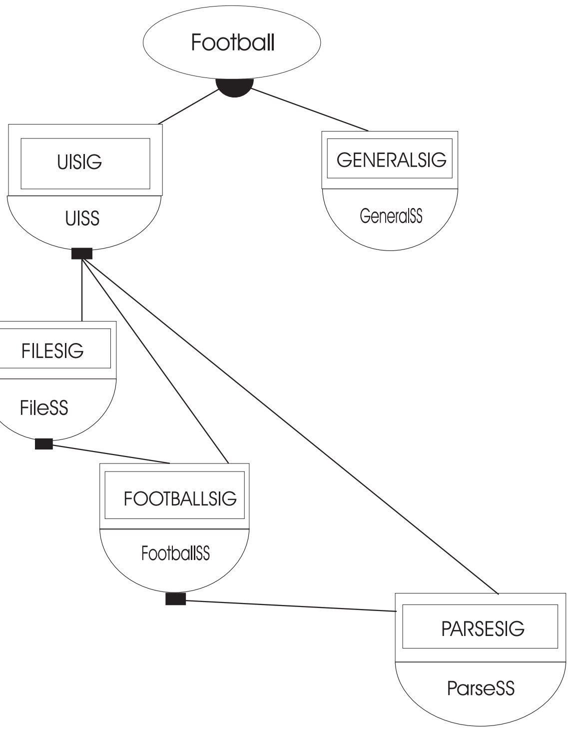

Figure 70: Updated Subsystem Dependency Diagram

signature. Thus there are also specifications of:

- the type `results`, which is used by `writeResFile` and `readResFile`;
- the function `readResults` , which is used by `readResFile`;
- the function `writeResults` , which is used by `writeResFile`;
- the type `teams` , which is used by `readTeamsFile`;
- the function `readTeams` , which is used by `readTeamsFile`;
- the type `leagueTable` , which is used by `appendLTToFile`;
- the function `writeLeagueTable` , which is used by `appendLTToFile`.

Therefore the signature `FOOTBALLSIG` currently mediates access to its associated subsystem for more than one client subsystem. However, it is clear that each requires access to a different collection of entities, which will eventually be reflected in separate exclusive signatures.

Since we encourage an approach built on information hiding if a type is specified in an exclusive signature it should not be accompanied by its constructor signature. Section 6.4 describes how one can model abstract data types in FAD. The specification of a type in an exclusive signature implies that entities of a client subsystem can be declared over the type, but the absence of a constructor signature signals that they have no knowledge of the construction of the type. Any intra-subsystem relationship does not require an entry in an exclusive signature but may later be categorised as an inter-module relationship and be specified in an exclusive signature that mediates access to a module. We describe type/function host analysis at the module level in Section 7.2.7.

Each subsystem's *subsystem description document* (SSDD) will be updated to record the collection of subsystems upon which it is dependent. Each subsystem is recorded with its associated exclusive signature. This is illustrated with the SSDD for `UISS` presented in Figure 71. The current version of `FOOTBALLSIG` is declared in an exclusive signature description document, which we present in Figure 72.

The development of exclusive signatures for each subsystem facilitates the assigning of subsystem development responsibilities to development units. Each unit will be responsible for one or more subsystems, but no two units have responsibility for the same subsystem. These assignments are recorded in new versions of the subsystem description documents.

Development of a Subsystem's 'used functions'

The development of each subsystem is the responsibility of a designated development team, which is recorded in the relevant subsystem description document. The development of a subsystem mirrors that of the whole system and should proceed in ignorance of the development of other subsystems, but in the knowledge of the interface presented by other used subsystems. One should begin by applying scenario analyses to the functions used by external users. The users in this case will typically be functions of other subsystems. Types used by the functions may need to be analysed simultaneously.

We illustrate this application of scenario analysis and type dependency analysis using the function `generateLT` of the subsystem `FootballSS`, which is used by the function `produceLT` of the subsystem `UISS` as represented in the function dependency diagram of

Subsystem Description Document	Football
Name:	`UISS`
Version:	19990721:1
Module(s):	
Exclusive Sigs:	
Subsystems Used:	`GeneralSS : GENERALSIG`
(with signature)	`FootballSS : FOOTBALLSIG`
	`ParseSS : PARSESIG`
	`FileSS : FILESIG`
Developed by:	
Description:	

This subsystem hosts the functions that implement the users' requirements. It also includes general purpose text-based I/O functions and may in future include entities that support other user interfaces.

Figure 71: Subsystem Description Document for the Subsystem `UISS`

Exclusive Signature Description Document	Football
Name:	`FOOTBALLSIG`
Version:	19990820:0
Subsystem:	
Type(s):	`results`, `teams`, `leagueTable`
Permissive sig(s):	

Function(s):	`createRes: parsedRes -> result`
	`inputResult:`
	`result -> results -> results`
	`checkResult: result -> bool`
	`readResults: string -> results`
	`writeResults: results -> string`
	`readTeams: string -> teams`
	`writeLeagueTable: leagueTable -> string`

Inherited Sig(s):	
Description:	

Interface to the subsystem `FootballSS` used by entities of the subsystems `UISS` and `FileSS`.

Figure 72: Exclusive Signature Description Document for the Signature `FOOTBALLSIG`

Figure 58. `generateLT` takes a value of type `teams` and returns a `leagueTable` value. The type `teams` is informally described as follows.

> A collection of football teams with an associated date that represents the last date of entry of information. Given the number of teams there is no requirement that they are stored in any particular order. Although support must be given for the retrieval, entry and updating of data there are no efficiency requirements.

The updated type dependency diagram for the type `teams` is presented in Figure 73. The diagram now reflects the assignment of entities to the subsystems of the system. The type is dependent on two types that will be declared in the utilities subsystem `GeneralSS`. The type `collection a` which may be an alias for a list type or some other container type, and the type `date`. The type is also dependent on the type `team`, which is declared in the same subsystem. The behavioural requirements of the type could be addressed at this stage but, reflecting the linear nature of the presentation, will be left to Section 7.3.1 when we discuss the development of permissive signatures.

The requirements of the function `generateLT` are presented below.

> The function is responsible for generating a league table from current team data. A league table entry must be generated for each team. The entry will include the team's name, its performance data home and away, and its total points. The team entries will be ordered first by total points, then by goal difference, goals scored and finally alphabetically.

Adopting a modular approach, the function `generateLT` can be described in terms of two other functions: a function that selects the required information from every team, `selectNamesAndData`, and another which generates a league table from this information, `createLT`. We describe the model of this scenario analysis in the function dependency diagram presented in Figure 74.

In common with the approach adopted earlier in Section 7.2.4 each subsystem will be developed as a collection of modules. Development of an initial module architecture both supports the principle of least commitment and furthers the development of a system based on information hiding.

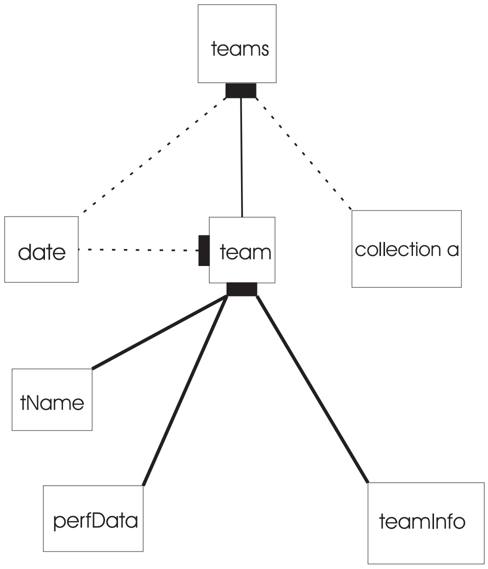

Figure 73: Updated Type Dependency Diagram for the Type `teams`

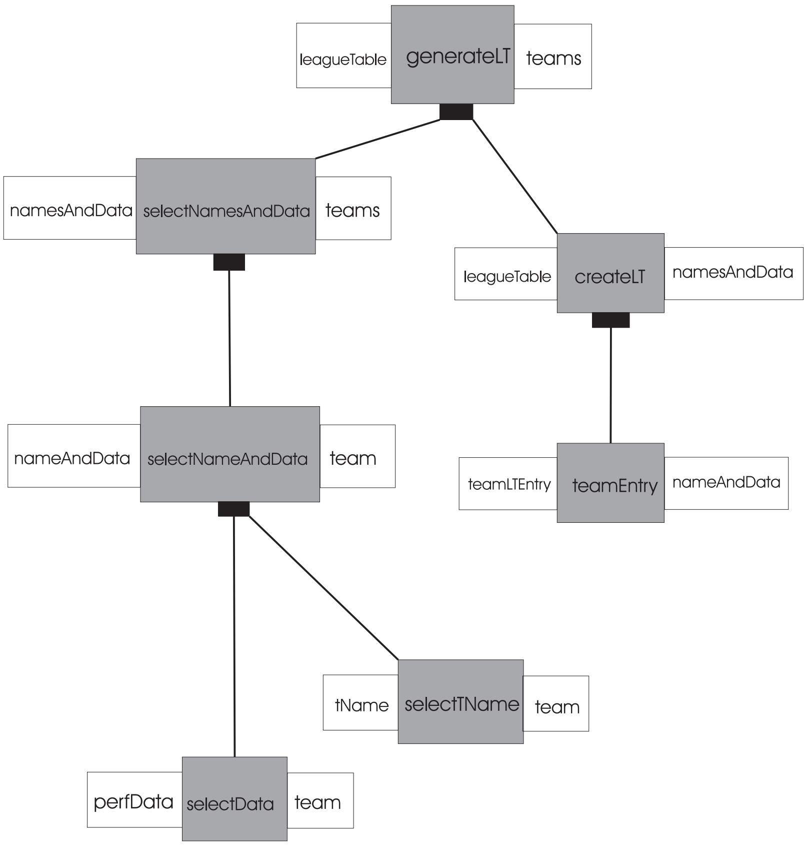

Figure 74: Function Dependency Diagram for `generateLT`

7.2.7 Module Architecture Analysis

The guiding principles of modularity applied at the system level are equally applicable at the subsystem level. That is, one should seek to develop independent cohesive units that are loosely coupled with other units. Module architecture analysis takes the description of a subsystem, its associated exclusive signatures and the results of scenario and type dependency analyses applied to 'use functions', and returns a model of the module architecture of the subsystem. The model is described through a collection of modules that are linked through module use relationships via their associated exclusive signatures.

The model will satisfy the modular development criterion through localizing 'intimate' knowledge requirements within each module. That is, if an entity requires knowledge of another entity's implementation then they are candidates for housing in the same module. If, however, the relationship is one where an entity only requires knowledge of the existence of another entity (and possibly some associated operations) then they can probably be declared in separate modules. For example, the standard libraries for Haskell 98 [101] are a collection of modules where a type is typically declared with a collection of functions that support behaviour over the type, and require intimate knowledge of the construction of the type.

Information hiding can be achieved by creating a module for each type specified in an exclusive signature associated with the subsystem. One then assigns the type and functions that implement behaviour over the type to the same module. A module may also include other types that are used by the signature type but are only of local use. For example, the type `perfData` that represents the performance data of a football team will be declared in the same module as the type `team`. The initial module architecture for the subsystem `FootballSS` has seven modules:

- `TeamsMod`, which hosts the type `teams` that represents a collection of football teams;

- `TeamMod`, the module housing the type `team`, which represents an individual football team. A football team has an unique name, performance information, and other team-specific data;

- `ResultsMod`, which hosts the type `results` that represents a collection of football

Function	Module	Comment
`generateLT`	`LeagueTableMod`	The function that generates a league table.
`selectNamesAndData`	`TeamsMod`	Selection function for `teams`.
`selectNameAndData`	`TeamMod`	Selection function for `team`.
`selectData`	`TeamMod`	Selection function for `team`.
`selectTName`	`TeamMod`	Selection function for `team`.
`createLT`	`LeagueTableMod`	The function that creates a league table.
`teamEntry`	`TeamMod`	Selection function for `team`.

Table 4: Function Host Analysis Related to the Function `generateLT`

results;

- `ResultMod`, the module housing the type `result` that represents a single football result;
- `PlayersMod`, which hosts the type that represents a collection of players, `players`;
- `PlayerMod`, the module housing the type `player` that represents a football player; and,
- `LeagueTableMod`, which hosts the type of league tables, `leagueTable`.

Once a set of modules have been declared one applies type/function host analysis to the micro unit entities of the subsystem. In this incarnation of the technique entities are being assigned to modules rather than subsystems. We present in Tables 4 and 5 the result of type/function host analysis applied to the entities in the function dependency diagram of Figure 74.

The function `generateLT` could be either assigned to the module `TeamsMod` or the module `LeagueTableMod` since it uses types declared in these modules. The function creates values of the type `leagueTable` and thus should be declared with the type. The function requires access to the implementation of the type `leagueTable`, where in contrast it has deferred such requirements of the type `teams` to the function `selectNamesAndData`. Hence the function was assigned to the module `LeagueTableMod`.

Type	Module	Comment
`teams`	`TeamsMod`	Host for type `teams`.
`team`	`TeamMod`	Host for type `team`.
`leagueTable`	`LeagueTableMod`	Host for type `leagueTable`.
`namesAndData`	`TeamsMod`	Type constructed from components of `teams`.
`nameAndData`	`TeamMod`	Type constructed from components of `team`.
`perfData`	`TeamMod`	Type used to construct values of `team`.
`tName`	`TeamMod`	Type used to construct values of `team`.
`teamLTEntry`	`TeamMod`	Values generated from values of `team`.

Table 5: Type Host Analysis Related to the Function `generateLT`

The type `team` has been assigned to a separate module from the type `teams` since the module `TeamsMod` should support the behaviour required of the type `teams` and not that of the type `team`. Any functions over the type `teams` that use functions over the type `team` should not require access to their implementation. The type `team` and its associated functions can therefore implement their behaviour using any design without affecting the dependencies. The reusability of components is significantly enhanced through this type of modular approach.

Once all entities have been assigned to a module one can update the various dependency diagrams to reflect the module architecture. The subsystem architecture resulted in the use relationships being categorised either as *inter-subsystem* or *intra-subsystem*. Now we further categorise the intra-subsystem relationships into either an *inter-module* relationship or an *intra-module* relationship. This is illustrated in Figure 75 where we present the updated function dependency diagram for the function `generateLT`.

The module `TeamMod` hosts the function `selectNameAndData` and the two functions upon which it depends as indicated by the thick use relationships connecting the functions. However, `createLT` is declared in the module `LeagueTableMod` and the function it uses `teamEntry` is declared in `TeamMod`. The function `generateLT` of the module `leagueTableMod` uses the argument type `teams` of a different module `teamsMod`, which is indicated by the vertical line on the function side of the type box.

Any functions that use a type through an inter-module relationship should be further investigated using scenario analysis. Abstraction will probably result in any such function depending on other functions declared in the type's module.

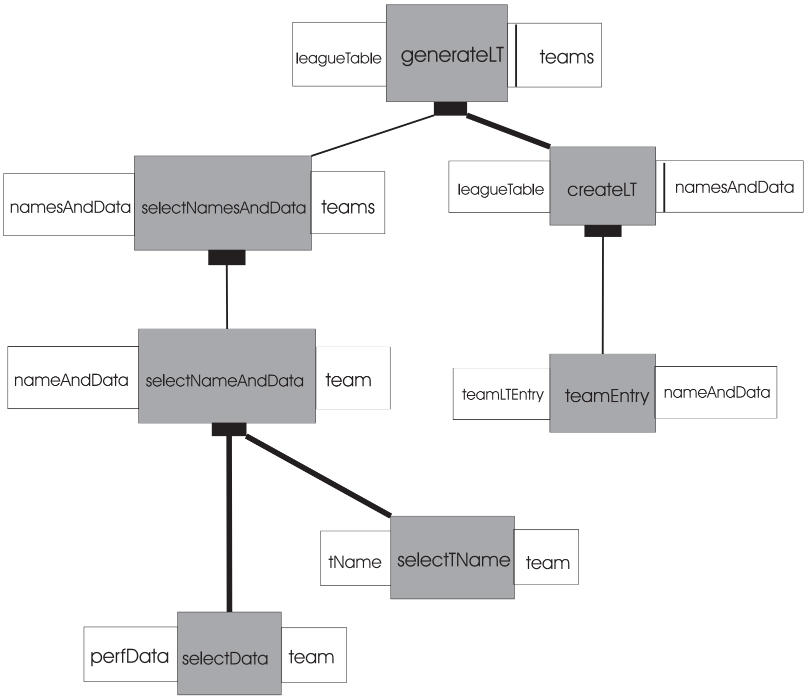

Figure 75: Updated Function Dependency Diagram for `generateLT`

Entities assigned to module **M** can use entities of module **N** of the same subsystem if and only if there is a *module use relationship* from **M** to **N** and the required entities are specified in the associated exclusive signature. We describe the development of an initial set of exclusive signatures in the following section. Upon completion one has a set of models that could be used to support the prototyping of a subsystem.

7.2.8 Module Exclusive Signature Analysis

Module exclusive signature analysis takes the results of the analyses described in the previous section, and the signatures associated with the host subsystem, and develops a collection of exclusive signatures through which a subsystem's modules are used. Every entity declared in an exclusive signature associated with the host subsystem must also be declared in at least one signature associated with a module of the subsystem. For example, the function `generateLT` of the subsystem `FootballSS` is used by the function `produceLT` of the subsystem `UISS`. It is therefore declared in the exclusive signature `FOOTBALLSIG` associated with the subsystem `FootballSS`. The function `generateLT` has been assigned to the module `LeagueTableMod` and therefore must be declared in

Entity	**Type Specification**	**Signature**
`createRes`	`parsedRes -> result`	`RESULTSIG`
`inputResult`	`result -> results -> results`	`RESULTSSIG`
`checkResult`	`result -> bool`	`RESULTSIG`
`readResults`	`string -> results`	`RESULTSSIG`
`writeResults`	`results -> string`	`RESULTSSIG`
`readTeams`	`string -> teams`	`TEAMSSIG`
`writeLeagueTable`	`leagueTable -> string`	`LTSIG`
`results`		`RESULTSSIG`
`teams`		`TEAMSSIG`
`leagueTable`		`LTSIG`
`generateLT`	`teams -> leagueTable`	`LTSIG`
`selectNamesAndTeams`	`teams -> namesAndData`	`TEAMSSIG`
`selectNameAndData`	`team -> nameAndData`	`TEAMSIG`
`teamEntry`	`nameAndData -> teamLTEntry`	`TEAMSIG`

Table 6: Entity Signature Specifications

the exclusive signature that links the *partition relationship* to the module. If one was implementing the system in Haskell this signature would typically be the export list provided by the module. All other signatures will be implemented as import lists when the module is used.

Any entity used by an entity declared in another module of the subsystem must be specified in the exclusive signature that mediates access for the relevant client module. Initially each module will be associated with a single exclusive signature and the *client-specific* signatures will be developed during the design phase of the methodology. This is described in Section 7.3.5.

We will illustrate module exclusive signature analysis through analysis of the subsystem exclusive signature `FOOTBALLSIG` - described in Figure 72 - and the results of the function/type host analysis applied to `generateLT`. Table 6 presents the results, where each function is recorded with its type specification and the signature in which it is specified. The module with which each signature is associated should be obvious from the signature's name.

The signature `RESULTSSIG` is recorded in the description document of Figure 76. The

Exclusive Signature Description Document	Football
Name:	`RESULTSSIG`
Version:	19990823:0
Subsystem:	`FootballSS`
Type(s):	`results`
Permissive sig(s):	
Function(s):	`inputResult:`
	`result -> results -> results`
	`readResults: string -> results`
	`writeResults: results -> string`
Inherited Sig(s):	
Description:	
Interface to the module `ResultsMod`.	

Figure 76: Exclusive Signature Description Document for the Signature `RESULTSSIG`

module architecture for the subsystem `FootballSS` is presented in a *module dependency diagram* in Figure 77. This is based on the analyses described thus far but will be iteratively developed as a result of further analyses.

The analysis phase is complete for the system (at least for this iteration) once a module architecture has been developed for each subsystem. With an incremental approach to development each subsystem can be developed at its own pace as long as milestones for the whole project are met. The design phase takes the deliverables of analysis and develops implementable designs of the macro and micro units. This will include further investigation of functions so that an efficient functional design can be modelled which uses, for example, polymorphism, overloading and higher-order functions. This involves taking advantage of existing entities recorded in the *data dictionary*. We describe FAD's data dictionary in Chapter 8. The following section describes the tasks and techniques of the design phase.

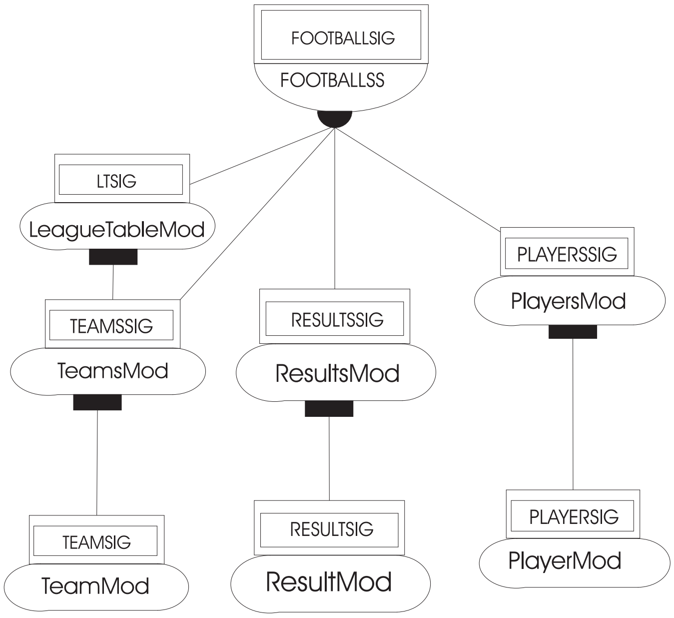

Figure 77: Module Architecture for `FootballSS`

7.3 Design

Design focuses on the delivery of a solution-domain focused model of the system. That is, where analysis is tied to the problem-domain albeit described in terms of the required paradigm, design aims to produce a model which can be implemented in as an efficient and effective manner as possible. However, it is clear that the importance of modularity, both in macro unit and micro unit development, has had a design impact within the analysis phase of development.

During the design phase, one takes the deliverables of the analysis phase and, using the various mechanisms provided by the paradigm, designs the various micro and macro units such that an efficient implementable design is returned. The transition from a largely analytical model to an implementable design is supported by the consistent paradigm-focus of the methodology and the fact that the diagrams and many of the techniques used during analysis are the same as those used during design. This also aids any iterative steps between phases or tasks within the phases. One can of course take the transition one step further and develop a model that reflects the idiosyncrasies of a particular implementation language.

During analysis OOADMs encourage the developer to build models of the system

Phase	Task	Techniques
Design	Design functions for purpose and reuse.	Scenario Analysis Permissive Signature Analysis Polymorphism/Overloading Design Higher-Order Design
	Type design.	Type Dependency Analysis Permissive Signature Analysis
	Design permissive and exclusive signatures.	Exclusive Signature Design Permissive Signature Design

Table 7: FAD Methodology – Design Phase

based on interacting objects. The design phase tends to focus on developing the internals of objects, introducing new classes that provide a controller rôle or some other implementation-specific rôle, and redrafting the inheritance hierarchy for efficiency reasons. For example, abstract classes are introduced to act as interfaces to several subclasses and generalization/specialization relationships are introduced where appropriate. One can also take advantage of the growing collection of reusable design patterns [49]. That is, one is looking to convert an analytical model that is drafted in terms of units of the OO paradigm into one that takes full advantage of the glue available to the OO developer.

With FAD one wants to take advantage of functional glue, which include parametric polymorphism and higher-order functions and the mechanisms available for the development of data types. An important part of design is the reuse of existing entities. We describe FAD's data dictionary and its support for reuse in Chapter 8. The deliverables of this phase aid the storage of entities in the data dictionary in a manner that improves the chances of reuse, and the discovery of potentially polymorphic, overloaded, or higher-order functions. This is simply achieved through adding to the key information that describes a function or type. The tasks and techniques of the phase are presented in a linear format as summarized in Table 7.

Architecture design is not included in Table 7. This is because the results of type and function designs will determine both the module architecture of subsystems and the subsystem architecture of the project. For example, if a type is implemented using a tree type, then use relationships between the relevant macro units will be declared and exclusive signatures introduced where necessary.

The initial focus of the design phase is function design. This task takes the current description of a function or collection of functions and further analyses them in terms of their behavioural requirements. The potential for polymorphism, overloading and the replacement of a collection of first-order functions with a single higher-order function are all reviewed.

Functions are the building blocks of functional software as described in Chapter 3. If the software implementers are provided with inadequate information upon which to implement the required functions then the software is likely to be inadequate itself. One cannot guarantee correctness through a FAD model, since the modelling language of FAD is not a formal specification language like Z [39] or VDM [151]. However, there is no reason why one can't support development within FAD with formal models written in a formal language. One can add formality to UML models through the object constraint language (OCL) [145].

Scenario analyses applied during the analysis phase deliver a collection of models that describe (to a certain level) the analysed functions. The analyses are applied until a set of exclusive signatures can be developed which reflect those entities of a macro unit which are used by clients. Thus functions that depend on other entities declared in the same module may not yet have been analysed. We encourage further analyses to be applied to such functions. For example, the function `teamEntry` that is used by the function `createLT` as illustrated in Figure 75, takes the name and performance data of each team and returns a league table entry. It uses two other functions declared in its module `TeamsMod` that generate the total points for a team and its goal difference. The function dependency diagram is presented in Figure 78.

A technique which provides further function development (and type development) information is *permissive signature analysis.*

7.3.1 Permissive Signature Analysis

The modelling language of FAD includes two types of signatures that were described in Sections 5.2.3 and 5.3.3. An exclusive signature presents to a client macro unit, exactly

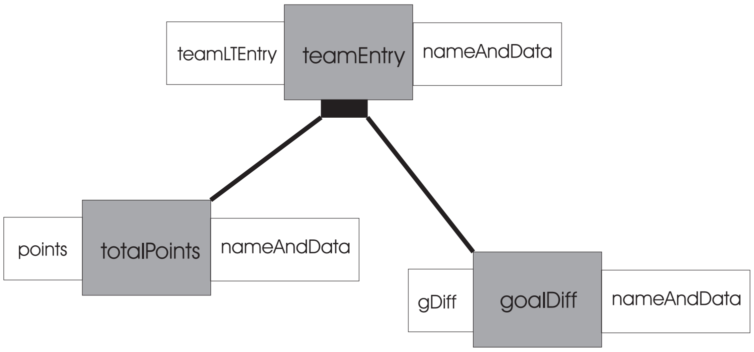

Figure 78: Function Dependency Diagram for the Function `teamEntry`

those entities that can be used from its associated macro unit. A permissive signature specifies some behaviour that is implemented over its associated type(s). That is, where an exclusive signature signals *only this*, a permissive signature indicates *at least this*. A permissive signature therefore makes behaviour explicit and specifies the entities that support the behaviour. Each permissive signature can be reused through association with another type whose type constructor is of the same *kind*. One can also create new signatures through *inheriting* the specifications of an existing signature as described in Section 5.4.7.

Permissive signature analysis takes a function and determines whether it requires its types to support any particular behaviour. The behaviour may be required over a type used by one of its types. If required, then one can either use an existing permissive signature that specifies such functionality or declare a new one. Existing permissive signatures are recorded in FAD's data dictionary and we will describe how they are categorised and the support for reuse in Chapter 8. The signature is then associated with the appropriate type in the function specification. The type is said to *instantiate* the permissive signature and this will be recorded in the type description document.

We present an example from the case study using the function `selectNamesAndData` of the module `TeamsMod`. The function is used by the league table generating function `generateLT` as modelled in Figure 75. The function is described in the FDD in Figure 79.

From the textual description of the function one can build an abstract model of the function's behaviour. The function applies a data extracting function to each item of

Function Description Document	Football
Name:	`selectNamesAndData`
Version:	19990810:1
Module:	`TeamsMod`
Arity:	1
Contract Association:	
Instantiations:	
Type Specification:	`teams -> namesAndData`
Functions Used:	`selectNameAndData`
Description:	
This function takes the collection of teams and returns the name and performance data of each team. Each team is selected and its name and performance data is retrieved.	

Figure 79: Function Description Document for `selectNamesAndData`

its collection-type argument. The function therefore requires the collection type used by the type `teams` to support the application of a function to each of its items. This can be modelled by associating the type `collection a` with the permissive signature `MAP` which specifies mapping functionality. The signature specifies the higher-order function `map`. We present the description document for `MAP` in Figure 80 and the updated function specification for the function `selectNamesAndData` in Figure 81.

The function dependency diagram in Figure 81 provides the developer with a range of information that includes:

- an abstraction of the function's main behaviour. This abstraction is reusable beyond its current application;
- the functions used to deliver the required functionality;
- the potential for the implementation of an overloaded function in a language which supports overloading. Section 7.3.2 describes how FAD supports the design of polymorphic and overloaded functions. If the implementation language does not

Permissive Signature Description Document Football

Name:	`MAP`
Version:	19990824:0
Module:	
Parameter(s)(kind):	`m :  * -> *`
Entities:	`map :`
(with type specs.)	`(a -> b) -> m a -> m b`
Inherited Signature(s):	
Description:	

This signature specifies mapping behaviour.

Figure 80: Permissive Signature Description Document for `MAP`

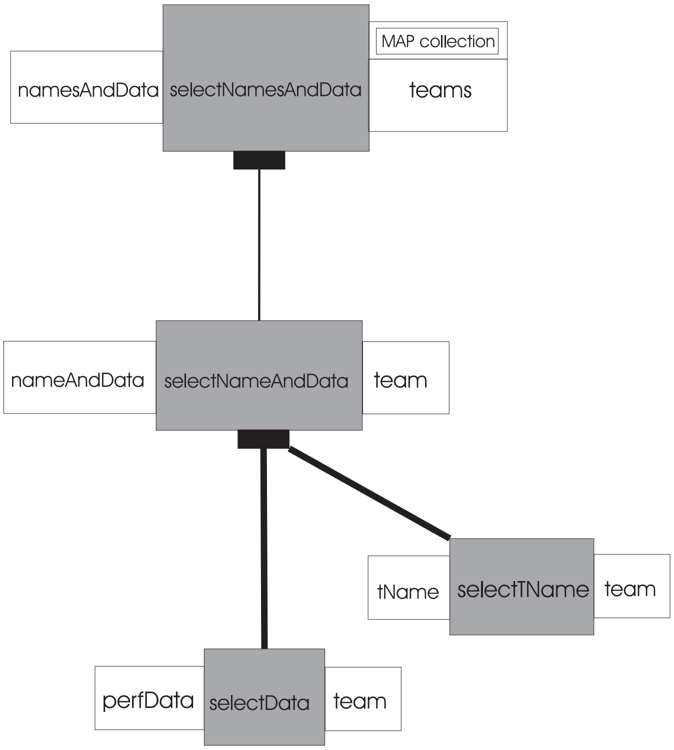

Figure 81: Updated Model for the Function `selectNamesAndData`

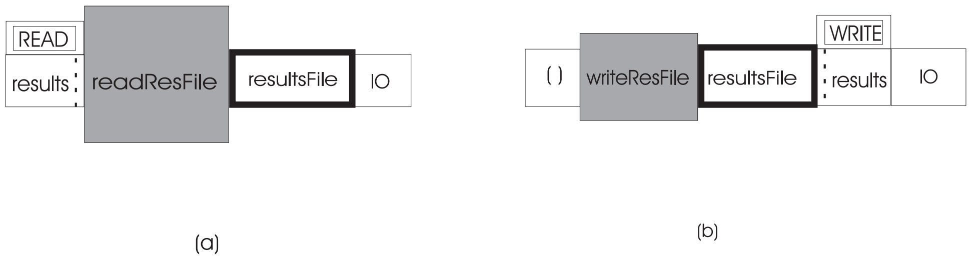

Figure 82: Updated Function Models for `readResFile` and `writeResFile`

support overloading then either unique or qualified names will be required for the functions that match those specified in a permissive signature;

- some guidance on the development of the type `collection a` which we expand on in Section 7.3.4.

For the second example we return to the functions `readResFile` and `writeResFile` that were first described in Section 7.2.5. They used the functions `readResults` and `writeResults` to implement the required 'readability' and 'writability' functionality over the type `results`. Permissive signatures provide an alternative means of describing the required functionality, with the benefit that the signature is reusable and can be associated with more than one type. We therefore introduce two permissive signatures `READ` and `WRITE` that include the specifications `read : string -> a` and `write : a -> string` respectively. We present the updated specifications for the functions `readResFile` and `writeResFile` in Figure 82.

Permissive signature analysis returns models of functions that include descriptions of behavioural abstractions. In the following section we describe how these models play an important rôle in the discovery of potentially polymorphic or overloaded functions.

7.3.2 Polymorphism/Overloading Design

Parametric polymorphism and constrained polymorphism (overloading) provide mechanisms for reuse in functional languages. Where parametric polymorphism supports the use of the same code over multiple types, constrained polymorphism supports the reusability of a name but not necessarily code. A description of polymorphism within the functional programming paradigm and how it compares to that of OO is provided in Chapter 3.

A polymorphic function can replace several monomorphic functions whose behaviour is exactly the same. For example, monomorphic identity functions over each type can be replaced by a single polymorphic function. Functions that return the length of a list of some monomorphic type can be replaced by a single polymorphic function that returns the length of any list. In both of these cases the set of monomorphic functions exhibit exactly the same behaviour, and are not reliant on any functionality being supported by their types.

In contrast constrained polymorphic functions do require some specified functionality to be supported either by their types, or some type(s) used by one or more of their types. Jones [66] motivates the argument in favour of type classes through examples of functions that sum two values of the same type and test the equality of two values of the same type. Monomorphism is too restrictive in both cases since in most functional languages there are several numeric types and even more types whose values can be tested for equality. However, a polymorphic function is inappropriate in both cases since there are non-numeric types that don't support, for example, arithmetic operators and some non-equality types such as the functional types.

The developer therefore needs support, both in the discovery of potentially (constrained) polymorphic functions and in the reuse of such existing functions. We leave the description of the latter process to Chapter 8. Permissive signatures, or the lack of, provide significant support in the development of (constrained) polymorphic functions. We suggest that the following guidelines should be followed.

- If a function is specified with types with no associated permissive signatures then the function could have a polymorphic type. This is because the function's types have no explicit required functionality, which suggests that the type's values do not influence the behaviour of the function. The identity function is an example of this type of a function;

- If a function's types have associated permissive signatures whose parameters are all of non-$*$ kind then it could have a polymorphic type. The values of the types used to construct an argument value are not required to support any particular functionality. The length function is an example of this type of function;

Permissive Signature Description Document	Football
Name:	`CONTAINER`
Version:	19990826:0
Module:	
Parameter(s):	`c :  * -> *`
Entities:	`add : a -> c a -> c a`
(with type specs.)	`remove : int -> c a -> c a`
	`find :`
	`(a -> bool) -> [a] -> maybe a`
Inherited Signature(s):	
Description:	
This signature specifies common functionality over container types.	

Figure 83: Permissive Signature Description Document for `CONTAINER`

- If the function is specified with at least one permissive signature then it could be declared as an overloaded function. Clearly this will require implementation language support for overloading. There is clearly an overlap with the above case illustrated by the length function that could be declared as an overloaded function. Another example is the function that sums two numeric values.

We will illustrate application of these guidelines with some examples from the case study. The I/O function `inpRes` uses the function `inputResult` to input a new result into the current collection of results (see Figure 60). Permissive signature analysis has resulted in the declaration of a new permissive signature with a parameter of kind `* -> *`, `CONTAINER`, which supports typical functionality of a container type such as the addition of a new item and the removal of an existing item. The signature is described in the permissive signature description document presented in Figure 83.

Three functions are specified that implement the addition of an item, the removal of an item in a specified position, and finding a value which satisfies a particular predicate. We have not included a function which removes all items matching an inputted value since this would require equality functionality of the items' type. The function

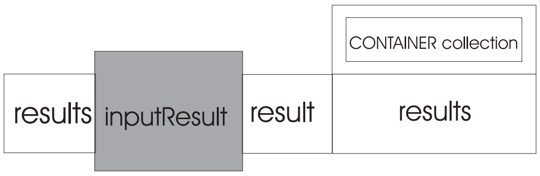

Figure 84: Potential Polymorphic or Overloaded Function

`inputResult` is modelled as in Figure 84. The type `collection a` is used to construct values of type `results`, which is fully described in Section 7.3.4.

The behaviour of `inputResult` does not require any behaviour over the type `result` that supplies the items contained in the collection. Thus the function could be defined as (or use) a polymorphic function over the type `collection a` or an overloaded function over any type that instantiates the permissive signature `CONTAINER`.

As a second example we return to the functions `readResFile` and `writeResFile` that are used by the function `inpRes` to read results from a file and write results to a file. The functions, which are modelled in Figures 82(a) and 82(b), require the type `results` to support the behaviour specified by the associated permissive signatures `READ` and `WRITE`. These signatures specify functions for reading and writing behaviour. Since `READ` and `WRITE` have parameters of kind * the functions could not be polymorphic but could possibly be implemented as overloaded functions.

A polymorphic function whose type includes unconstrained type variables must be universally accessible and thus declared in a module in the subsystem `GeneralSS`. Polymorphic functions that are defined over constructed types should be assigned to the same module as the type. For example, functions that are declared over any list should be assigned to the module that hosts the list type. Overloaded functions that are specified in a permissive signature will be declared in the module that hosts the type that is associated with the signature. Other constrained polymorphic functions are declared in a module in the subsystem `GeneralSS`.

In the following section we describe how permissive signatures can signal the potential for development of higher-order functions.

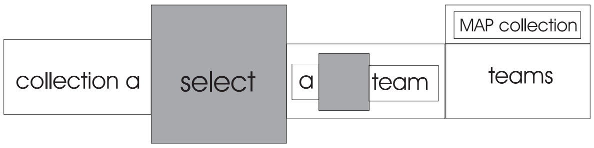

Figure 85: The Higher-Order Function `select`

7.3.3 Higher-Order Function Design

FAD supports the modelling of multiple argument functions in their *curried* and uncurried form. With the curried form, new functions can be created through the *partial application* of the functions to an incomplete set of arguments. In the following chapter we describe how entities are stored in the data dictionary and how this supports the potential for function creation through partial application.

In this section we describe how FAD supports the development of functions with functional arguments. Higher-order functions capture a common pattern of computation across several functions. Thus one is able to replace several first-order functions with a single higher-order function. In each case the function is applied to a functional argument which was previously used in the body of the first-order function. Permissive signatures can be used to highlight common patterns of computation. For example, the function `selectNamesAndData` described in Figure 81, applies the function `selectNameAndData` to each value of type `team` in a collection of teams. The pattern of computation is made explicit by the association of the permissive signature `MAP` with the unary type constructor `collection` used by the type `team`. We could replace `selectNamesAndData` with a higher-order function `select` that takes a functional first argument as described in Figure 85. The model of the function `generateLT` that previously used `selectNamesAndData` requires updating as illustrated by the function dependency diagram of Figure 86.

Of course not all higher-order functions are so easily discovered. Two functions may use a function of the type `t1 -> t2` but without any explicit behavioural requirement beyond the application of the used function to an argument. If the functions have similar models then they may exhibit common abstractions. That is, if their *function dependency diagrams* present common patterns then there is the possibility of a common abstraction. Common model patterns could indicate common abstractions, which may

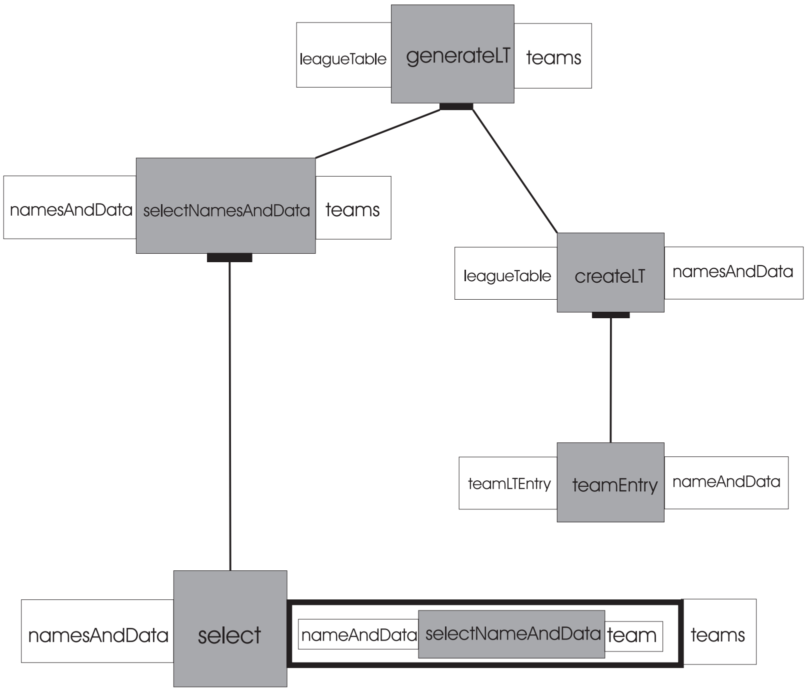

Figure 86: Updated Version of the Function `generateLT`

result in some efficiency in design. Although not currently a part of FAD, one could look to record particular model patterns to support reuse of design and the discovery of common abstractions. Design patterns are an interesting area of future research within the functional programming community. They are already practised within OO development [113, 47, 49, 27, 18].

In conclusion, FAD provides significant support for function development. This includes modelling a function as a collection of functions upon which it depends and providing support through permissive signatures for the development of polymorphic, overloaded, and higher-order functions. In Chapter 8 we describe FAD's data dictionary and its support for reusing existing functions and developing functions in parallel. The next section describes the task of type design.

7.3.4 Type Design

During the analysis phase, scenario analyses and type dependency analyses are practised in parallel in order to provide the information necessary to effectively specify a function.

In common with functions, types are investigated until every use relationship is an intra-module one. Some types might therefore require further type dependency analysis in advance of implementation.

Each non-basic type should be modelled in a type dependency diagram. In addition, permissive signature analysis makes explicit the behaviour that must be definable over a type. For example, the signatures `MAP` and `FOLD` indicate particular patterns of computation over any instantiating type, and `EQ` and `ORD` signal an equality type and ordered type respectively. Thus far permissive signatures have been associated with types in response to a behavioural requirement of a function. During type design one can take each type and determine whether any further permissive signatures should be associated with the type or any types upon which it is dependent. Types can then be developed that either use existing types which instantiate the permissive signatures or require the declaration of new signature instantiations.

We illustrate the results of further analysis with a detailed model of the type `results`. Its type dependency diagram is presented in Figure 87.

Thus the type `results` must be declared using a type that instantiates the permissive signature `CONTAINER`, and the `date` type. For example, using Haskell notation, one could implement the type as a product type as follows:

```
data Results = Rs Date [Result]
```

where the type `collection a` has been implemented as a list. The list type has the required `CONTAINER` functionality. We describe in the following chapter how one can match a type in development against an existing type.

Type designs may have an impact on the subsystem architecture, and module architecture of subsystems. For example, `ResultsMod` will now use `ListMod`, the module that hosts the list types and their associated operations. Module architecture design is therefore intimately linked to the design of types.

A value of the type `result` has four components: a `date` value, `homeTeam` and `awayTeam` values (which are implemented identically), and an `attendance` value. Once the design of a type is confirmed a constructor signature can be declared and associated with the type. Here is a possible implementation for `result`.

```
data Result = R Date (HomeTeam,AwayTeam,Attendance)
```

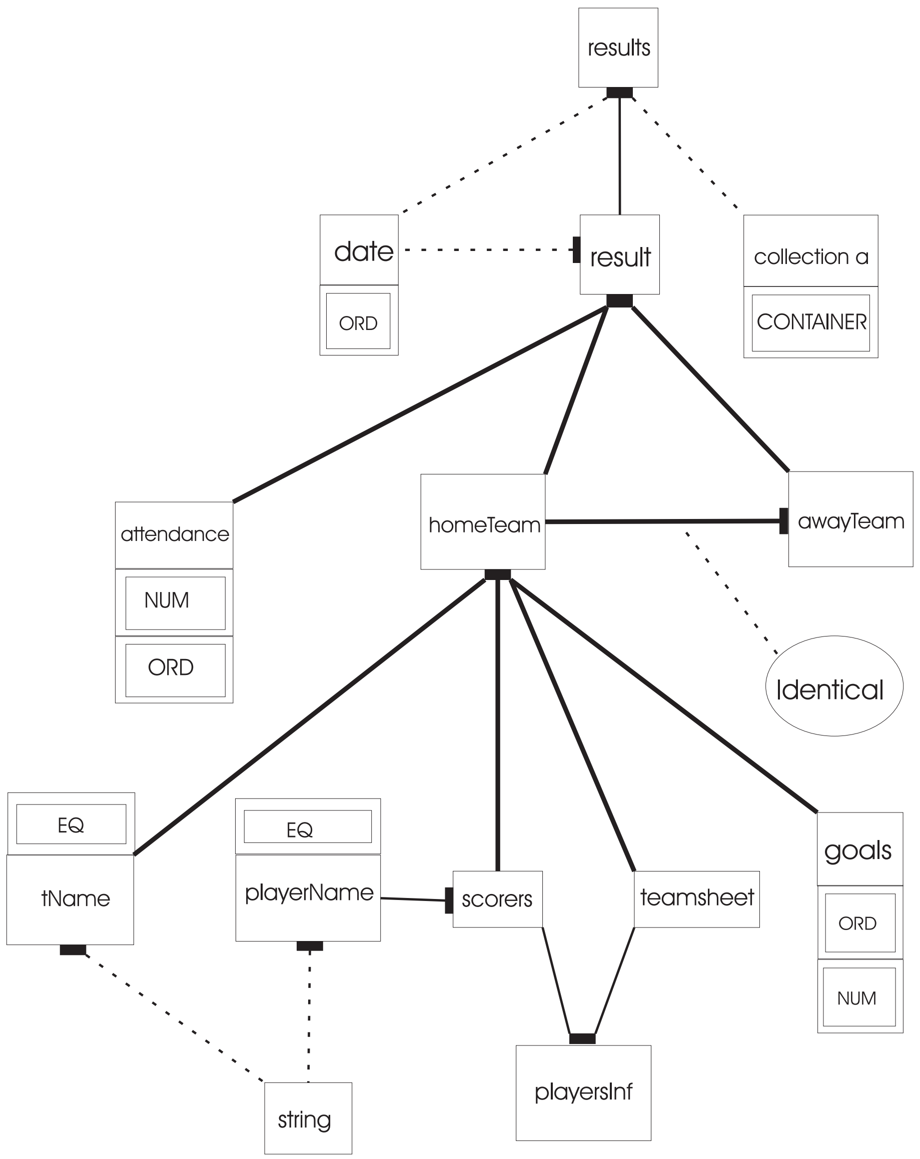

Figure 87: A Model of the Type `results`

Since the types `results` and `result` are declared in separate modules and the type `result` is abstract, one could change this implementation with any changes restricted to the module `ResultMod` which houses the type `result`. The `date` type can be implemented as any appropriate type that instantiates the permissive signature `ORD`, which specifies an ordering functionality over its instantiating types. Since the signature `ORD` inherits the signature `EQ`, any instantiating type must also have equality functionality. Details regarding permissive signature design are described in Section 7.3.6.

7.3.5 Exclusive Signature Design

Exclusive signature design takes the current set of exclusive signatures (which are typically one-one mapped with a subsystem or module) and designs a set of signatures that state the exact interface presented to each client of a module or subsystem. During the analysis phase exclusive signatures provide a specification for macro unit developers and a guide to the functions (and their types) available for use from other macro units. Subsystem exclusive signatures provide input into the development of exclusive signatures associated with their modules. One now needs to provide a truer reflection of the interaction between macro units. That is, the signature associated with a unit may be redeclared as a collection of signatures each mediating access to the unit for a different client.

For example, the module architecture for `FootballSS` presented in Figure 77 is updated to that presented in Figure 88. The only change is that the signature `RESULTSSIG` has now been redesigned as three signatures that provide the exact interface required by the client. Details of two of the signatures are presented in Figure 89.

7.3.6 Permissive Signature Design

Every permissive signature is recorded in a description document. In Chapter 8 we describe the approach to storing permissive signatures on the basis of the number and kind of their parameters.

Permissive signatures are used to declare a behavioural requirement over a type. To avoid potential confusion a permissive signature should specify only that which is required. That is, if a function requires *mapping* behaviour over a type then the associated permissive signature should specify only that behaviour. Through signature

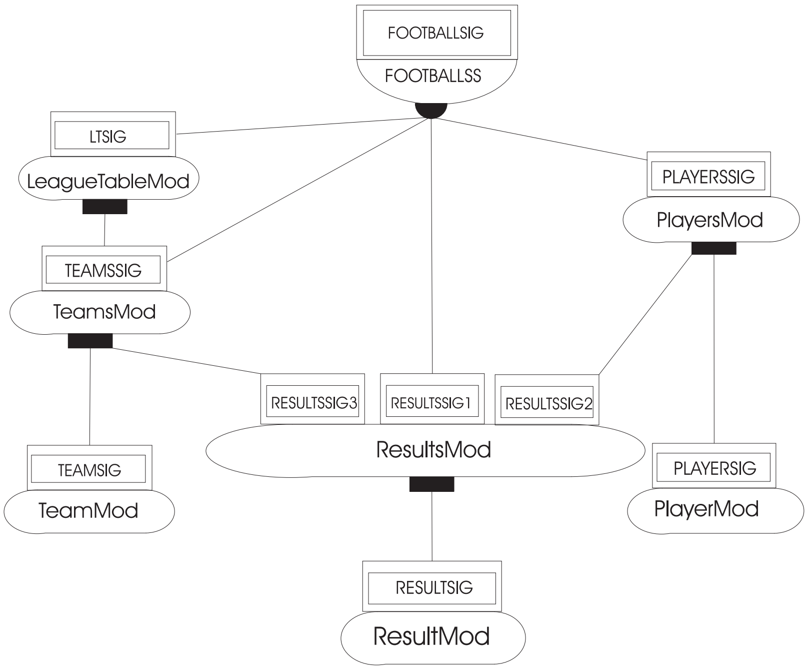

Figure 88: Updated Exclusive Signature Design for Modules of `FootballSS`

Figure 89: Exclusive Signatures Associated with the Module `ResultsMod`

inheritance one can develop signatures that specify a range of behaviour. However, one should not develop a signature through inheritance unless the resulting collection of behaviour is actually required. That is, one should err on the side of caution, and not tie signatures unnecessarily to an inheritance hierarchy.

For example, the signature `ORD` that specifies functions that implement ordering over a type, is an extension of one that supports equality, `EQ`. Thus one can declare `ORD` by inheriting the specifications of `EQ` and adding other required specifications.

7.4 Summary

In this chapter we have presented the methodology of FAD by describing its tasks and the techniques used to implement a task. The techniques deliver models described in terms of the modelling language of FAD. The methodology is neither intended to reinvent good practice in functional programming nor prevent bad practice, as was the motivation for the introduction of structured programming and its associated analysis and design methodologies. Rather FAD should support software development in the functional programming paradigm by plugging the hole due to the lack of paradigm-specific methodologies. FAD's modelling language and techniques support good practice rather than encouraging a new approach to building systems within the paradigm.

The final element of the methodology is its data dictionary. The following chapter presents an overview of FAD's data dictionary, how it supports the reuse of existing entities, and the design of entities in development.

Chapter 8

Data Dictionary

One of the benefits of using an ADM as a tool in software development outlined in Section 4.2.4, is that it provides significant support for documenting development. This has several uses of which we highlight two of the most significant. Firstly, it provides a record of development for future reference either during maintenance or as an input into the development of a new system. Secondly, it can provide excellent support during development especially in relation to discovering common abstractions and reusable entities. Of course implementation code provides its own form of documentation, but this is only available when the code is written. Unsurprisingly it presents a picture of the idiosyncrasies of a particular language rather than a clear statement of a system's design and functionality.

With large projects developed by multiple units there is a danger of substantial duplication of effort. An ADM with a supporting CASE tool can reduce this risk both through recording entities and designs in an efficient manner, and providing mechanisms for reuse and the discovery of common abstractions in existing entities and entities in development.

In Section 8.2 we describe FAD's data dictionary. We describe how each type of unit is stored and how this supports the requirements stated above. In the following section we review related work on matching entities in development to existing entities.

8.1 Related Work

Most of the research within the functional programming community on supporting reuse has focused on matching functions in development to those defined in a library. The matching *key* in most cases is the function's *type signature*. Therefore, the matching criterion is syntactic and not semantic.

Runciman and Toyn [122] describe an approach where the function in development may not have an explicit type signature. They present techniques for developing a *key type* for the new function, which can be compared against the types of existing functions. One major limiting factor of their approach is that it enforces an ordering on the arguments. That is, although the types `a -> b -> [b]` and `Int -> Char -> [Char]` match, the type `Char -> Int -> [Char]` will not match the latter type. Several reusable functions will be missed due to this constraint.

Rittri [117] removed the restriction on the order of a function's arguments and developed a process where one could match a query type against an isomorphic type, where the isomorphisms are the ones that hold in all cartesian closed categories. Rittri enforces the explicit declaration of a query type but only allows exact matches up to isomorphism. Thus, for example, a monomorphic type will not match a polymorphic type. Once again potential matches may be missed due to this constraint.

Zaremski and Wing developed two approaches to matching modules as well as functions. They have a syntactic approach called *Signature matching* [154] which matches on types, and a semantic approach called *Specification matching* [155], which matches formal specifications of the behavioural characteristics of functions and modules. Since formal methods are beyond the scope of FAD, we will only review signature matching.

Zaremski and Wing define a collection of *basic matches* of function signatures that can be combined to produce other matches. Modules are matched on the basis of their signatures using these basic matches. The basic matches are:

exact match: two signatures are equal up to variable names and user-defined type names;

generalised match: the query type exact matches an instance of the library component type;

specialized match: the library component type is an exact match of an instance of the query type;

unify match: the two types have common instances that match exactly;

uncurry: the uncurried versions of the two types are exact matches;

reorder match: a reordering of the library component type is an exact match for the query type.

A signature matcher has been implemented in SML and integrated into the author's local SML programming environment. However, the onus is on the user to determine the appropriate matches to apply. This is not a trivial task since some *relaxed* (non-exact) matches may result in far too many functions and an exact match in too few. There are no metrics which measure the most efficient route to a successful match.

Park and Ramjisingh [94] take a significantly different approach to those described above. They argue that an efficiently organised component library would maximise the potential for reuse. They describe an approach to the storage of functions where functions are grouped through their arity. Intra-group functions are linked through *type-substitution* and inter-group functions are linked through *argument-substitution*. That is, two functions **f1** and **f2** of the same group are linked if the type of **f1** is more general than the type of **f2**. Alternatively one can say that the type of **f2** is an *instance* of the type of **f1**. The type of **f2** can therefore be created through substituting one or more types for type variables in the type of **f1**.

Two functions **f3** and **f4** of different groups are linked if the one of lower arity has a type that is an instance of the type of the function of higher arity with one or more arguments removed. That is, the type is an *applicative type instance* of the higher arity type. A query type can therefore be matched against the same type, a more general type, a more specific type, or a type with more arguments, which can be made an instance of the query type once some arguments are removed. However, matching is constrained by the order of the arguments.

An and Park [4] have taken grouping a step further and removed the emphasis on the order of arguments. Thus functions are assigned to function groups based on their arity, and within each function group is a collection of extended set types. For example, the extended set type `{int,char} -> bool` includes the types `int -> char -> bool`, `char -> int -> bool`, `(int,char) -> bool`, and `(char,int) -> bool`. That is, each extended set type is a collection of isomorphic types as described by Rittri [116]. A node is created for each set type. Intra-group links are now between two nodes within the same group and inter-group links between two nodes in different groups. The links are defined as in Park and Ramjisingh [94]. Hence one benefits both from having a structured repository of components and access to isomorphic functions within a node.

Although this section is titled *Related Work* the work on matching components has focused on matching entities - typically functions - in development with functions defined in libraries. The matching requirements for a methodology are more varied. Matching with existing entities is still required, but so are matching entities being developed with similar behavioural requirements and matching non-function entities such as types and signatures.

In the following section we describe FAD's data dictionary and how it provides an efficiently organised approach to entity storage and satisfies the above requirements.

8.2 FAD Data Dictionary

FAD's data dictionary is a medium for the storage of the collection of description documents for all the declared micro and macro entities. We describe in the following sections the criteria for placement of each form of entity. Those entities that are not described in a section are simply stored alphabetically. Each system entity will be described by one or more description documents that provide an historical record of development of the entity. The information recorded will include descriptions of any changes and the reasons for the changes. We describe in the following sections the storage of the set of description documents for each entity, but will use the latest version to determine its storage situation. That is, as entities are developed they may be repositioned within the data dictionary. For example, a type may be associated with a permissive signature when previously it had no such association. This will change where it is stored as described in Section 8.2.2.

8.2.1 Functions

Each function is recorded in a series of *function description documents*. The description includes: the function's arity, type specification and associations between argument and result types and permissive signatures. These are the important entries when determining the storage location of the function and links between functions.

Functions are stored using the following criteria, which are applied in the enumerated order.

1. Function arity.
2. Associated permissive signature kind.
3. Alphabetical.

Functions are initially grouped by their arity. That is, we have adopted Park and Ramjisingh's approach of grouping all functions with a single argument together, all functions with two arguments together and so on.

We then assign the functions in each group to a subgroup of functions whose types instantiate a permissive signature of a specified *kind*. A permissive signature's kind is recorded in its description document. All functions which require a type/signature association of kind `*` are grouped together. Functions which require the instantiation of a permissive signature of kind `* -> *` but not any of kind `*` are grouped together and so on. Finally, any functions which do not require the instantiation of any signature are grouped together. Within each of the subgroups the functions are stored alphabetically. For example, the function `inputResult` (see Figure 84) will be grouped with the functions of arity 2, with a permissive signature of kind `* -> *`. Thus if one wants to develop a function that takes two arguments and has mapping behaviour, one can look in this group.

In contrast to the matching of functions with implicitly or explicitly declared types, during development a function may use types that simply have a name and some association with permissive signatures. This approach to organising functions will place these functions with other functions with similar behavioural requirements.

Functions with the same arity and permissive signature associations will therefore be stored in the same group. This enhances the chances of discovering potential polymorphic functions and overloaded functions. It also reduces the likelihood of identical definitions being bound to two different functions of the same type. In addition, if one wants a function of arity **n** with a behavioural requirement specified by a permissive signature of kind `* -> *`, then one may find a function in the subgroup of arity **n+1** and permissive signature association of the same kind that could create the function through *partial application*.

Finally, if one wants a function over a type **t** then one can initially search in the function's arity/permissive signature subgroup, and if unsuccessful, can then review the module that hosts the type. Since systems are built on information hiding, functions that implement behaviour over the type should be declared in the module that hosts the type.

8.2.2 Types

A type is recorded in a series of *type description documents*. The description includes the kind of the type's constructor and any permissive signature associations. The types are categorised using the following criteria applied in the enumerated order.

1. Type constructor kind.

2. Permissive signature instantiation.

3. Alphabetical.

A type is initially assigned to a group on the basis of the kind of its constructor. Thus all types with nullary type constructors will be grouped together, as will all types with unary type constructors. Within each of these groups the types are multiply assigned to the subgroup of types that instantiate a specified permissive signature. However, if a type instantiates several signatures which are related through inheritance, then it is only assigned to the signature which permits the most behaviour. Within each of these groups the types are stored alphabetically.

Thus if one wants to find a type that instanitates the permissive signature `ORD` one only has one place to look. This reduces the chances of repetition of type definition and increases the likelihood of reuse.

8.2.3 Permissive Signatures

A permissive signature is recorded in a series of *permissive signature description documents*. The description includes a listing of parameters and their kind. Permissive signatures are categorised using the following criteria applied in the order enumerated.

1. Number of parameters.

2. Kind of parameters.

3. Alphabetical.

Each permissive signature is assigned to a group on the basis of their number of parameters. The signatures `NUM`, `ORD`, `FOLD`, and `MAP` each have one parameter and will therefore be grouped together. Within each group, signatures with a parameter of kind `*` are assigned to a subgroup. The remaining signatures with a parameter of kind `* -> *` are assigned to another subgroup and so on. Within each subgroup the signatures are stored alphabetically. Thus `NUM` and `ORD` are assigned to the same group, as are `MAP` and `FOLD`.

If one is developing a signature with a single parameter of kind `* -> *` then one can look in the appropriate group and determine if an acceptable one exists, or if one could be created that extends an existing one through inheritance. Alternatively, the new signature could be extended to create an existing signature.

8.3 Summary

We have outlined in this chapter how the FAD data dictionary provides an organised repository for defined elements and elements in development. The criteria for storing each element were described. Organised storage increases the likelihood of reuse and the discovery of common abstractions.

Chapter 9

Summary

In this thesis, we have presented arguments in favour of an analysis and design methodology which supports software development within the functional programming paradigm. We presented evidence of significant support for object-oriented development and the general benefits of including a methodology within the process of software development.

Popular methodologies, such as the Booch Method and SSADM, are underpinned by a graphical modelling language which delivers abstract models of software designs. They are not however visual programming languages since they deal with abstractions rather than implementation details. We believe that a methodology whose language has elements in harmony with the functional programming paradigm and whose techniques encourage and support the development of functional designs is required. We cannot prove, in any formal and rigorous sense, that applying the methodology actually improves the efficiency with which one develops software, or the effectiveness of the implemented solution. However, we can offer software developers a packaged approach to development where the media used allow focus on the *essential complexity* of software development, whilst avoiding the *accidental complexity* inevitable when switching paradigms.

In the appendix to this thesis we applied FAD to the development of a consistency checker for a CASE tool. Its support for the building blocks and glue of the functional paradigm enforced an approach that was consistent with the paradigm from the initial stages through to design. We list below the specific successes of the application followed by the modifications/additions that we believe will enhance the modelling language and methodology.

We claim the following successes:

- the notation was easy to use, unambiguous and presented the models in a clear and readable manner. Other notations have embedded a function's arguments and return values within the function notation. We believe presenting types external to their associated function - as first described in Section 5.2.2 - provides several benefits. These include:
  - a function's type specification is clear;
  - it emphasises the importance of types during development; and,
  - it allows behavioural requirements to be associated with the types in a clear and explicit form.
- the multiple views of a system supported by FAD deliver clear, focused models uncluttered by unnecessary information;
- the adoption of a single set of diagrams naturally supported the iterative development of models throughout development. Models tend to require updating rather than replacement;
- permissive signatures (see Section 5.2.3) are an important element of the modelling language. They allow behavioural requirements to be added to type information in a form that is independent of any type and thus reusable across types of the appropriate kind. They can be naturally implemented as types classes in implementation languages that offer such support as described in Section 6.6;
- independent macro unit and interface model elements. This proved invaluable during development where one wants to be able to specify an interface to a module that is appropriate for a particular relationship. For example, in the appendix we have developed three exclusive signatures that provide interfaces to the module `StateMod`. Each satisfies a particular abstraction requirement. Full details of this example are provided in Section A.6;
- an initially type-centric approach to module development and interaction supports the discovery of the functions that exist over a type. In some cases one may have a

choice of modules which could host a function, but it will still minimize the search space;

- delaying the implementation details of a type in favour of specifying the behavioural requirements and types used, encourages an approach to development in which one is not tied prematurely to a particular set of implementations. That is, we have adopted the *principle of least commitment*, which requires as much abstraction as possible in order to minimize the scope of future implementation decisions. This was illustrated, for example, in the development of the type `components` (see Section A.6.4) and the various substate types.

We also believe that in light of our experiences with the case study there are areas of the modelling language and methodology that could benefit from modification and extension. We list these below:

- there is a need for a 'shorthand' notation for an interface that specifies everything in its associated module or all but a few of the hosted units. This also applies where an interface specifies everything hosted by a module used by its associated module;

- a review of 'case' notation. That is, where a function has input-specific behaviour we currently present each alternative in a separate diagram although typically in the same model. This can result in a lot of component repetition and is thus somewhat inefficient. Other modelling languages have adopted an approach where one presents the various cases on a single diagram, which although more efficient, can result in a less readable model;

- the case study did not address any of what Peyton Jones has described as the *awkward squad* [99]. FAD currently supports development using pure functional programming languages. It will require extension to support the various means of interacting with the external world.

9.1 Summary of Contributions

The major contribution of this thesis is a methodology for developing functional software. Although popular within other paradigms this development medium has been hitherto absent from the functional programming paradigm.

We claim the following particular contributions:

1. A modelling language for building abstract models of functional designs. The syntax and semantics of the language were described informally as is common with modelling languages.

2. A collection of integrated techniques which takes the deliverables of requirements engineering and return software design that is best implemented in a functional language.

3. A set of documentation which provide a medium for recording system entities and presenting a history of design decisions. Each document includes entries which guide the storage of the document in the data dictionary. The data dictionary is an organised repository for storing entities. It supports the reuse of existing entities and the discovery of common abstractions between entities.

4. A case study that provides evidence of the suitability of FAD in a functional software development process.

9.2 Future Research

There are several areas of future research that would be of clear benefit in the application of FAD.

There is a need for a CASE tool that supports the application of FAD. A methodology without a CASE tool is like a programming language without a compiler. Developers are attracted to methodologies through their CASE tools, and thus, future research must focus on the development of a CASE tool for FAD. FAD provides no guidelines for *consistency-checking* and *version control*. This is not unique to FAD since it is uncommon for a methodology to provide (non-generic) details on how consistency-checking or version control can be practised. One can of course use the documented material to manually check for consistency of design, and manage version control, but this could soon become unwieldy. CASE tools typically provide support for consistency checking models developed using their associated methodology. Thus any CASE tool would need to support consistency-checking. The case study presented in the appendix could be used as part of this research.

Design patterns are increasingly popular within the OO community. The scope and usefulness of such patterns with functional designs is an interesting area of research. A functional modelling language could be used to describe reusable abstract designs and possibly to uncover common abstractions in existing designs.

Appendix A

Analysis and Design of a Consistency Checker

In this appendix we present a significant example of the application of FAD. FAD is best applied through a CASE tool that will support *inter alia* the recording of units in development and the checking of the consistency of the various models that together describe a design. It is the CASE tool's *consistency checker* that is the focus of this application. It will be developed as one of the subsystems of the CASE tool project.

In the following section we provide a description of a consistency checker that includes a definition of an *inconsistency*, and in Section A.2 there is a detailed overview of the requirements of the consistency checker. Sections A.3 and A.4 present a representative selection of scenario and type dependency analyses that span the major issues regarding the development of the functionality of the consistency checker. In Section A.5 we analyse the module architecture of the subsystem where the modules, exclusive signatures and module use relationships required by the consistency checker are developed. Design issues are discussed and illustrated in Section A.6, and a summary of the development and a brief overview of work to be done are given in Section A.7.

FAD, in common with most ADMs, provides multiple views of a system in development. This is one of the major benefits of their application. However, multiple views can lead to inconsistencies between the views, and these inconsistencies may be very difficult to discover if the system is of a non-trivial kind. Thus most CASE tools provide a means of resolving such problems in the form of a *consistency checker*.

A.1 Consistency Checker

A consistency checker is a significant part of a CASE tool. Although a modelling language supports the delivery of a design for a system one cannot assume that the design is consistent. That is, one cannot assume that the design is implementable. This is particularly true when designing a large system that may be represented in a series of models. This is precisely the case when using FAD where one is encouraged to develop models that provide various views of the system in development. A visual scan of such models is unlikely to discover potential inconsistencies either within a model or between models. A consistency checker is the tool that enables a methodical approach to the discovery of design inconsistencies. In addition, the incremental and iterative approach to development encouraged by FAD, can only be practised effectively if one has a mechanism for controlling the introduction of new elements, and the replacement of existing elements in an updated design.

Here we are using the term *model* as an identified collection of *elements* of the modelling language. An *element* is any micro unit, macro unit or relationship of FAD. Thus, for example, a model could be a module dependency diagram or a function dependency diagram, a mixture of both, or simply a collection of unrelated elements. Since one is building a system with the intention of future implementation, it is necessary to build one that can be implemented. An *inconsistency* is something that cannot be implemented. We illustrate an inconsistent design with an example. In **Model 1**, the function `aFun` uses the function `bFun`. In **Model 2**, `aFun` is hosted by module `AMod` and `bFun` is hosted by module `BMod`. In **Model 3**, `BMod` uses `AMod` via the exclusive signature `ASIG` but there is no module use relationship in the other direction. Figures 90(a), 90(b), and 90(c) present a graphical representation of these three models.

An inconsistency exists between the dependence of `aFun` on `bFun` and the lack of a module use relationship from `AMod` to `BMod`. Thus any implementation of this design would include an error due to the lack of visibility of `bFun` from `aFun`. For example, in Hugs 98, if the module `AMod` is declared in the file `AMod.hs` and `BMod` is declared in `BMod.hs` then the following error occurs:

```
ERROR "AMod.hs": Undefined variable "bFun"
```

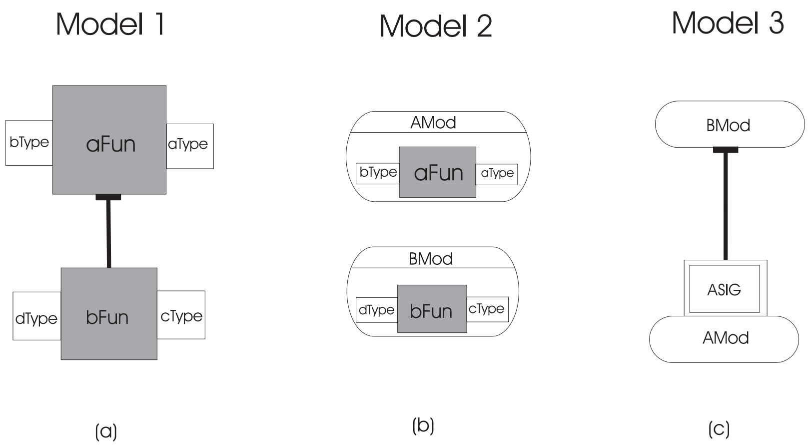

Figure 90: An Example of Inconsistency

A consistency checker should report the above inconsistency thus allowing the developers to resolve the problem pre-implementation. However, a consistency checker neither provides solutions to any problems nor reports on poor or inefficient design. It may highlight potential areas of concern but its primary rôle is to determine whether a design based on the models of development is consistent and thus implementable. This is analogous to the program error-spotting rôle played by a compiler.

In the following section we present the requirements of a consistency checker. These will provide the basis for the development of the checker.

A.2 Requirements Analysis

We present in this section a list of identified requirements each accompanied by some commentary. Each requirement is a consistency check. However they can be further categorised as either *pass/fail* checks or *warning* checks. A *pass/fail* check must be passed. The failure of such a check signals an inconsistency. A *warning* check discovers an aspect of a design which may result in an inconsistency, but either because of the limitations of a consistency checker or the variability in implementation languages, one cannot guarantee that it is an inconsistency.

Many of the *pass/fail* checks rely on one unit being *visible from* another. This is a non-symmetrical relationship that we define as follows:

A micro unit **B** is *visible from* the micro unit **A** if precisely one of the following is true:

- either **A** or **B** is not associated with a host module. During the early stages of development micro units may be introduced without a host module. The default is that such units are visible from any other unit and vice versa. This is to avoid unwanted consistency checking failure due to an incomplete design;
- **A** and **B** are hosted by the same module;
- **B** is hosted by a module **BMod** in the same subsystem as the module **AMod** that hosts **A**. There is either a module use relationship from **AMod** to **BMod** with **B** specified in the mediating exclusive signature, or there is a path from **AMod** to **BMod** via one or more intermediate modules where each module use relationship linking the modules is mediated by an exclusive signature that specifies **B**;
- **B** is hosted by a module **BMod** hosted by a subsystem **BS** that is used by the subsystem that hosts the module in which **A** is declared. **B** must be specified in the exclusive signature that mediates use of the subsystem, and in the exclusive signature that mediates the partition relationship between the subsystem **BS** and **BMod**, or a module that is linked to **BMod** via a path as described in the case above. This is illustrated in Figure 91, where to aid readability, we have limited the specifications presented in the exclusive signatures to those required for the example.

A module **M** is *visible from* a module **N** if precisely one of the following holds:

- either **M** or **N** is not hosted by a subsystem for the same reasons given above; or
- modules **M** and **N** are hosted by the same subsystem.

The *pass/fail* checks are:

Model Consistency: a model must be consistent relative to existing models. The

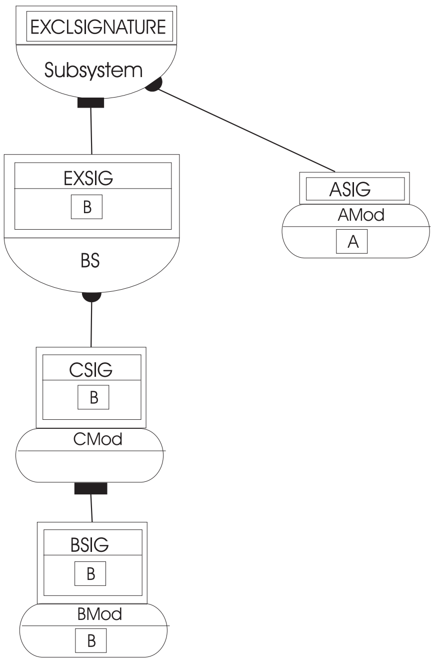

Figure 91: Illustration of *visible from* Relationship

consistency of a model will depend on the consistency of its elements. This is fully described in Section A.3.1.

Function Argument and Result Types: the types which provide the argument or result values of a function must be *visible from* the function.

Function Use: all functions used by a function must be *visible from* the function.

Type Use: all types used by a type must be *visible from* the type.

Module Use: a module may only use a module which is either hosted in the same subsystem or if either is unassigned to a subsystem. That is, module **M** may only use module **N** if **N** is *visible from* **M**. A module is hosted in a unique subsystem for a given project. It may be assigned to another subsystem in a different project.

Exclusive Signature Mediation 1: a module/exclusive signature association must be consistent. This is true if precisely one of the following holds for each micro unit specified in the exclusive signature:

- it is hosted by the associated module;

- it is specified in an exclusive signature that mediates access to a module used by the associated module and this module/exclusive signature association is consistent.

Exclusive Signature Mediation 2: a subsystem/exclusive signature association must be consistent. This is true if precisely one of the following holds for each micro unit specified in the exclusive signature:

- it is hosted by a module **M** hosted by the subsystem and is specified in the exclusive signature that mediates the partition relationship with **M**;
- it is hosted by a module which itself is hosted by a subsystem used by the subsystem, and is specified in the mediating exclusive signature and the previous rule holds for the used subsystem.

Permissive Signature Instantiation: a type/permissive signature association must be consistent. This holds if:

- the permissive signature is visible from the type(s);
- for each parameter of the permissive signature there is an associated type whose type constructor is of the same *kind*. Section 5.4.4 provides details of the instantiation of a permissive signature by one or more types;
- for each micro unit specified in the signature a micro unit exists of the type required by the signature.

Constrained Polymorphism: a function that includes a type/permissive signature association must be consistent. This holds if:

- the function argument types and result type are *visible from* the function;
- the permissive signature instantiations exist and are consistent. That is, the instantiation must have been previously declared;
- the type(s) associated with each permissive signature are *visible from* the relevant argument or result type. That is, the type with which the permissive signature is associated must either be the type to which it is (graphically) juxtaposed or a type used by this type. We present an illustrative example

in Figure 92 where the permissive signature `EQ` is instantiated by the type `aType` that is used by the type `bType`. This is a consistent design.

Permissive Signature Inheritance: a permissive signature inheritance relationship must be consistent. This holds if:

- the inheriting signature has a parameter (or parameters) of the same kind as the parameter(s) of the inherited signature.

Uniqueness: this includes:

- uniqueness of type constructor names. Each type must have a unique name, which will be the type constructor name if it takes no arguments, or the type constructor name plus associated parameters (type variables or types) for non-nullary constructors. A type constructor name must begin with a lower case letter;
- uniqueness of permissive signature and exclusive signature names. These names must use only upper case letters;
- uniqueness of module and subsystem names. These names must begin with an upper case letter;
- each micro unit hosted by a single module;
- each module hosted by a single subsystem;
- a micro unit specified in at most one permissive signature up to inheritance;
- each macro unit use relationship must be unique. For example, if `AMod` uses `BMod` then there must be a unique exclusive signature which mediates this usage.

but does not include:

- uniqueness of function names. Since polymorphism - constrained and unconstrained - is encouraged by the methodology, the reuse of function names must be allowed. However, they should only be reused where there is potential for one of the forms of polymorphism. That is, if two functions of different arity share the same name then this is an inconsistency. This is

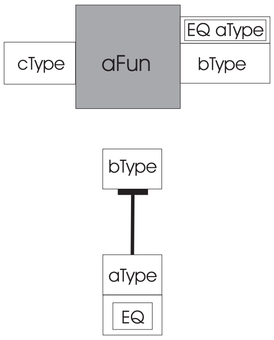

Figure 92: Constrained Polymorphism Example

because current functional languages do not support this form of function name overloading. Since a consistency checker is not a type checker one cannot perform the matching algorithms required to confirm the matching of types. Thus one can check for arity matching but not for type matching. However, one can report when a function name has been reused and leave it to the user to decide on the appropriate course of action. That is, this check uses a *pass/fail* check and a *warning* check. Arity matching is a pass/fail check and function name reuse is a warning check.

New Host Checks: these are a collection of checks that are triggered when micro units of existing models are assigned to a module, or a module of an existing model is assigned to a subsystem in the new model. Elements of existing models need to be rechecked since previously consistent designs may now be inconsistent. For example, a type use relationship may now be inconsistent if the related types are hosted in different non-related modules.

Update Checks: these are checks that are triggered when a model has been updated and may cause a previously consistent design to become inconsistent.

The following checks are *warning* checks or use *warning* checks.

Abstraction: if there is an abstraction barrier between a function and each of its types, and the function only uses functions that are not operations of the abstract type(s), then the user should be warned of the potential for the breakage of abstraction. Although this is not an *inconsistency* since it is perfectly valid for an abstract type to be an argument of a function that is not an operation of the type, and thus can be implemented in most functional languages, there is the potential for the abstraction barrier to be broken in the implementation of the function. The user should be advised of this type of design so that a decision can be made regarding the appropriate action.

We present an illustrative example in Figure 93. The function `aFun` uses the argument types `aType` and `bType` that are abstract relative to the function, since they are hosted in used modules and are specified in the mediating exclusive signatures `ASIG` and `BSIG` without their constructor signatures. `aFun` uses the functions `usedFun1` and `usedFun2`, neither of which is hosted with either of the types. Thus abstraction is potentially violated. See Section 6.4 for full details of FAD's support for abstract types.

Argument and Return Values: a function can be applied (partially or not) to values of the appropriate type and/or return a value of the appropriate type. Since a consistency checker is not a type checker one cannot confirm that a value matches the required type. However, if a type has known values one can do a matching on values. Also, if the type is a functional type one can check the arity of the function value against that of the functional type. Thus one can provide information for the user regarding the appropriateness of the value(s) used. The user receives a warning if any of the following situations occurs:

- a value is not a known value of the specified type;
- a function value's arity does not match that of the specified type.

Recursive Dependencies: any recursive dependencies are reported. This requires the investigation of each set of use relationships. For example, if a module **M** uses a module **N**, which itself uses module **M**, this is reported since the design may be non-implementable in some languages, and furthermore, it may indicate a poor module architecture design.

Each of the requirements listed above can be described as a function. For example, we have the function `functionUseCheck` that checks for the consistency of a function use relationship against the existing set of elements. These functions provide the foundation

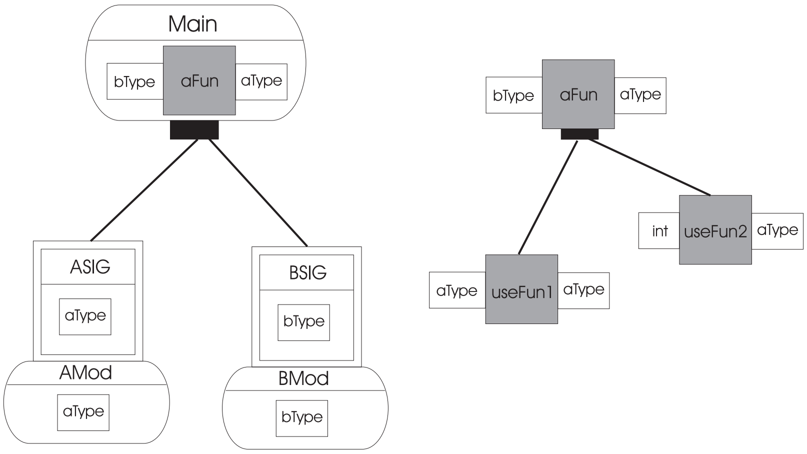

Figure 93: Abstraction Example

upon which a system will be analysed. Each function will be analysed in regard to both its type and behavioural requirements. The aggregation of these analyses should be the main functions and types required to implement the system. `functionUseCheck` is analysed in Section A.4.1.

We proceed in the following section with a selection of analyses of the functions that deliver the requirements outlined in this section.

A.3 Scenario and Type Dependency Analyses

When applying the scenario analyses one has to appreciate the inter-dependency between types and functions. How one develops types will have a direct impact on function development and vice versa. We will therefore present a mixture of scenario and type dependency analyses that will highlight the interplay between these techniques. The first scenario that we will investigate is that of checking the consistency of a model since the other requirements are subordinate to this one.

We have previously outlined how a system developed using FAD can be described by a collection of models. We will take a model-based and incremental approach to consistency checking. That is, rather than trying to compare a collection of models, as each model is *submitted* it is checked against existing models that have satisfied the consistency checker. Model submission is the process of adding the model to the current collection of system models. The aggregation of the elements of the existing models is used as the basis for the determination of the consistency of the submitted model. This also applies to the checking of a model which is an update of an existing model.

We will adopt an approach in which we present an informal description of an analysis followed by a description of the development of a FAD model. The informal description will typically provide a significant input into the description presented in the *Description Documents* for the units being analysed.

A.3.1 Consistency of a Model

Informal Description

The consistency of a model is tested relative to the aggregate of existing models. That is, one does not practice pairwise comparisons between the new model and each of the existing models but instead compares the design described by the new model against that described collectively by the existing models. We call this information the *state* of the system. A model is consistent if and only if each of its elements is consistent when checked against the state. It is therefore inconsistent if any of its elements introduce an inconsistency. The onus is therefore on the new (or updated) model to be consistent relative to the existing design and not on the existing design to change in order to accommodate the new model. However, inconsistencies can be introduced into the state due to new hosting relationships or a model being updated.

The consistency of an element of a model will also depend on those elements of the model which have already been checked. That is, one needs to update the information against which the model is being checked as the check is being processed. For example, if a model introduces a new type dependency diagram with some new types, then the types will be checked first. If these checks succeed then the types are added to the state against which the type use relationships are checked.

The manner in which a check proceeds depends on whether a model is *new* or an *update* of an existing model. If new, then one checks the model against the existing state. If an update, then the state requires some modification before checking. That is, since the model is replacing an existing model, some of the elements of the existing version may no longer be part of the state. This depends on whether they are part of any other existing model or are reused in the updated version of the model. If either or

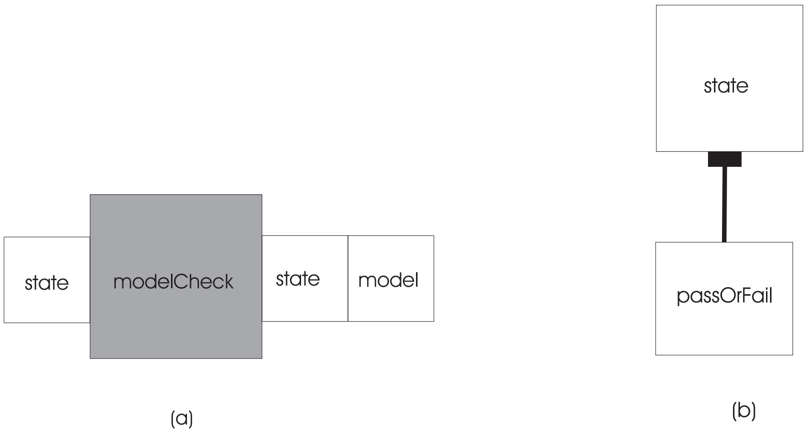

Figure 94: `modelCheck` function and the type `state`

both of these situations hold then they remain, and if not then they should be removed.

FAD Description

The function `modelCheck`, which checks the consistency of a model, takes two arguments of type `model` and `state`. The type `model` has as values the FAD models, and `state` includes the aggregation of existing elements. `modelCheck` returns a value of type `state` since it not only checks for consistency but updates the state for future checks. The type `state` uses the type `passOrFail` whose values reflect whether the check has been successful or not, and provides supporting information. The FAD graphical representation of `modelCheck` is presented in Figure 94(a) and a preliminary design for the type `state` is presented in Figure 94(b).

The function needs to determine whether the model is *new* or an *update*. Referring now to Figure 95, the type `model` must be an *equality type* whose equality is determined through its identifier. It therefore uses the type `modelID` that uniquely identifies each model and is also an *equality type*. Using the function `isIn`, one may test whether the model is *new*, and if so, one proceeds with the check of a new model using `newModelCheck`. Since one is checking for the existence of a model, the function `modelCheck` needs access to existing models either within the type `state` or as a separate type. We have decided to include this within the type `state` since this information will need to be updated upon the successful completion of the check.

If the model fails the *new* test - if `isIn` returns `True` - which implies that the model is an update of an existing model, then the `state` value requires modification using the function `modifyState`, and the function `oldModelCheck` is applied to the `state` value that is returned. These two alternatives are given in Figure 95 and a *Function Description Document* for `modelCheck` is provided in Figure 96.

`newModelCheck` and `oldModelCheck` have similar behavioural requirements. They both scan the elements of the `model` value being checked and will terminate the check if any element check fails, and will update the `state` value as each check succeeds. Thus one must be able to apply a consistency check to any element value. That is, a function `elementCheck` must exist over the `element` type and also over any type used by this type that represents the different units and relationships of FAD. These types, such as `function` and `typeUseRel` will be used either directly by the type `element` or via types used by this type. Details of the design of the type `element` are left to later in the development process and analyses of `newModelCheck` and `oldModelCheck` are presented in Sections A.3.3 and A.3.4.

At this point it is worth analysing the types `state` and `model`

A.3.2 The Types `state` and `model`

Informal Description

The type `state` plays a central rôle in the design of the consistency checker. It acts as a repository for the elements of existing models, a recorder of the identities of existing models and an indicator of the success or failure of the most recent check with additional information for the user. It is the `state` value that will provide the information against which a model is checked for consistency, and the information that determines if a model is new or is replacing an existing model. It is important therefore that one can add, remove and find elements, and similarly add, remove and find model identifiers.

If one is updating an existing model then the state requires modification in advance of the consistency check. However, if the check fails one wants to be able to return the state to its pre-modification form. This implies a design where one has three substates:

- one that records the aggregation of elements of existing models;
- one that records the elements of the model being checked that have passed their check. These elements are used together with those of the first substate in the

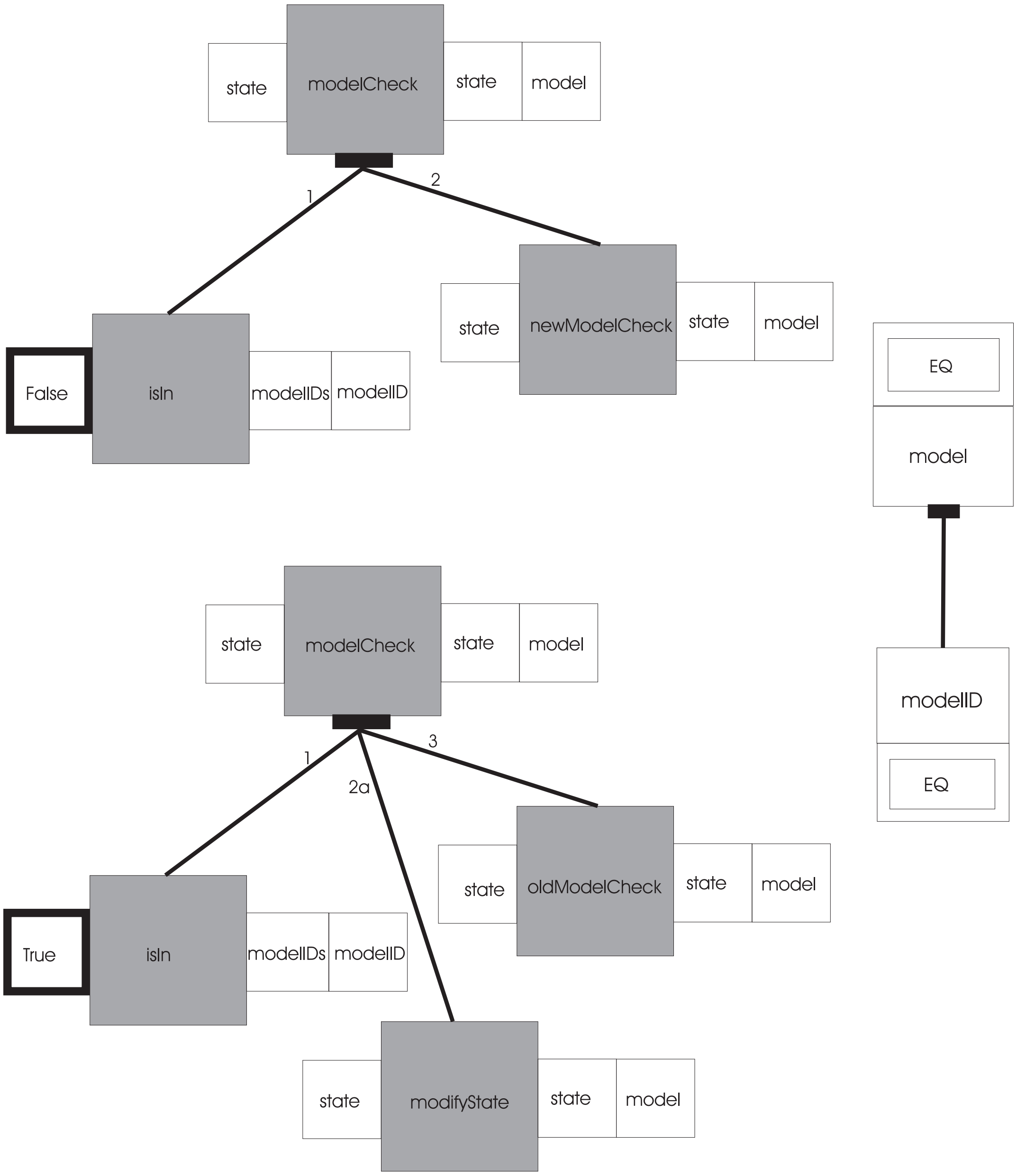

Figure 95: Conditional Behaviour of `modelCheck` and Design of `model`

Function Description Document CASE

Name:	`modelCheck`
Version:	20000710:0
Module:	
Arity:	2
Type Specification:	`state -> model -> state`
Contract Association:	
Instantiations:	

Functions Used:	`isIn : modelIDs -> modelID -> bool`
	`newModelCheck : state -> model -> state`
	`modifyState : state -> model -> state`
	`oldModelCheck : state -> model -> state`

Description:

The consistency of a model is tested relative to the aggregate of existing models. That is, one does not practice pairwise comparisons between the new model and each of the existing models but instead compares the design described by the new model against that described collectively by the existing models. We call this information the *state* of the system. A model is consistent if and only if each of its elements is consistent. It is therefore inconsistent if any of its elements introduce an inconsistency. In addition, the consistency of an element in a model will also depend on those elements of the model which have already been checked. That is, one needs to update the information against which the model is being checked as the check is being processed.

The manner in which a check proceeds depends on whether a model is *new* or an *update* of an existing model. If new, then one checks the model against the existing state. If an update, then the state requires some modification before checking. That is, since the model is replacing an existing model, the elements of the existing version may no longer be part of the state. This depends on whether they are part of any other existing model or are reused in the updated version of the model. If either or both of these situations hold then they remain, and if not then they should be removed.

Figure 96: Function Description Document for the Function `modelCheck`

checking of future elements of the model. If the model check terminates successfully then these elements are added to those of the first substate. If the check is unsuccessful then the first substate is left unchanged; and,

- one that records the elements that are (temporarily) removed from the first substate when the state is modified in advance of checking a model that is an update of an existing model. These are the elements that *only* exist in the previous version of the model. If the model check terminates successfully then these elements are discarded since they no longer exist in the design of the system. If the model check is unsuccessful then these elements are returned to the first substate since the previous version of the model remains in existence. Full details of the behavioural requirements when updating a model are presented in the analysis in Section A.3.4.

A model is an identified collection of elements. Each element can appear in one or more models and must be checkable for consistency against the state.

FAD Description

Here we are referring to the *Type Description Document* presented in Figure 97 and the type dependency diagram in Figure 98. The type `state` uses the following five types:

- `modelIDs`, which is the type of a collection of model identifiers;
- `subState1`, which is the type of existing model elements;
- `subState2`, which is the type of elements that satisfy checks during a model check;
- `subState3`, which is the type of elements that exist only in the previous version of a model for which an update is being checked; and,
- `passOrFail`, which signals success or failure of a check with supporting information.

`subState1`, `subState2` and `subState3` make use of the type `elements`, which uses values of type `element`. These three types may eventually be replaced by a single type that provides three fields of the type `state`. However, by treating them as separate types one has the flexibility either to implement them differently or decide to unify them into

Type Description Document CASE

Constructor Name: `state`
Version: 20000710:0
Kind: *
Module:
Types Used: `modelIDs`, `subState1`, `subState2`
`subState3`, `passOrFail`
Parameters:
Permissive sigs.:
Description:

The `state` value provides the information against which a model is checked for consistency and the information which determines if a model is new or is replacing an existing model. It is important therefore that one can add, remove and find elements, and similarly add, remove and find model identifiers.
If one is updating an existing model then the state requires modification in advance of the consistency check. However, if the check fails one wants to be able to return the state to its pre-modification form. This implies a design where one has three substates:
- one which records the aggregation of elements of existing models;
- one which records the elements of the model being checked that have passed their check. These elements are used in together with those of the first substate in the checking of future elements of the model. If the model check terminates successfully then these elements are added to those of the first substate. If the check is unsuccessful then the first substate is left unchanged; and,
- one which records the elements which are (temporarily) removed from the first substate when the state is modified in advance of checking a model which is an update of an existing model. These are the elements that *only* exist in the previous version of the model. If the model check terminates successfully then these elements are discarded since they no longer exist in the design of the system. If the model check is unsuccessful then these elements are returned to the first substate since the previous version of the model remains in existence.

Figure 97: Type Description Document for the Type `state`

a single type. One is therefore not forced into an early design decision. The type `elements` and `modelIDs` must use a collection type which instantiates the permissive signature `CONTAINER`. This signature specifies the functions `add`, `remove`, `empty` and `isIn` which guarantee:

- the ability to add an item to a collection;
- the ability to remove an item from a collection;
- an empty value for the collection; and,
- the testing for the existence of an item in the collection.

Both `remove` and `isIn` require the item type to be an equality type since in both cases they depend on the matching of an item with one in the collection.

We now refer to Figure 99 and to the *Permissive Signature Description Document* presented in Figure 100. Since each `element` value needs to be checked for consistency and the `state` value reflects the cumulative result of the application of the checks, the collection type used by `elements` must also support the *folding* of a function into the collection of values. This is guaranteed by the permissive signature `FOLD`, which we associate with the collection type. In addition, each element type must instantiate the permissive signature `CHECKABLE` that specifies the function `elementCheck`.

The type `model` uses two types:

- the equality type `modelID` whose value uniquely identifies a model; and,
- `elements` which is the type of the elements of the model.

We will continue in the next section with an analysis of checking the consistency of a new model.

A.3.3 Checking a New Model

Informal Description

A new model is checked against the existing set of models by checking each element of the model. As each element passes a check it is added to the state against which future checks are applied. If any element check fails then the model check fails and the details

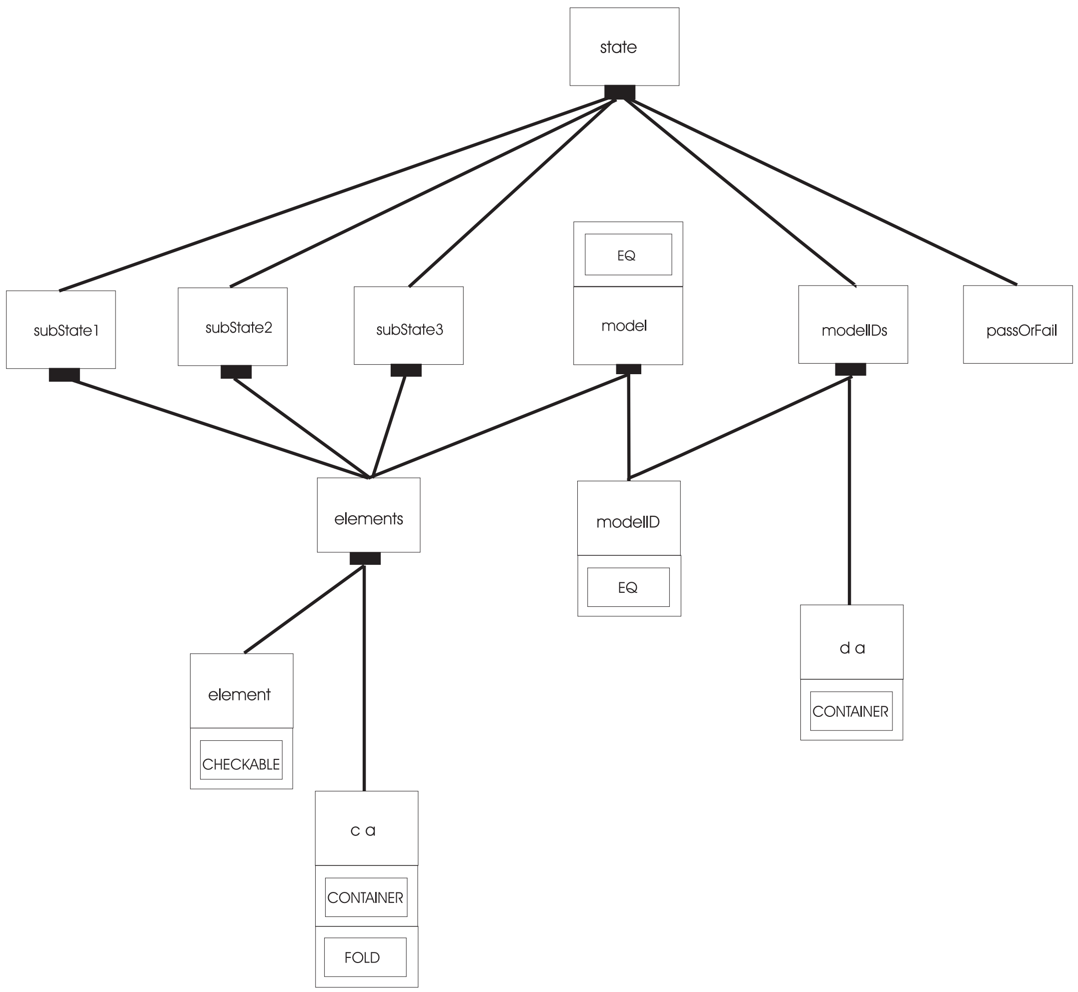

Figure 98: The Types `state` and `model`

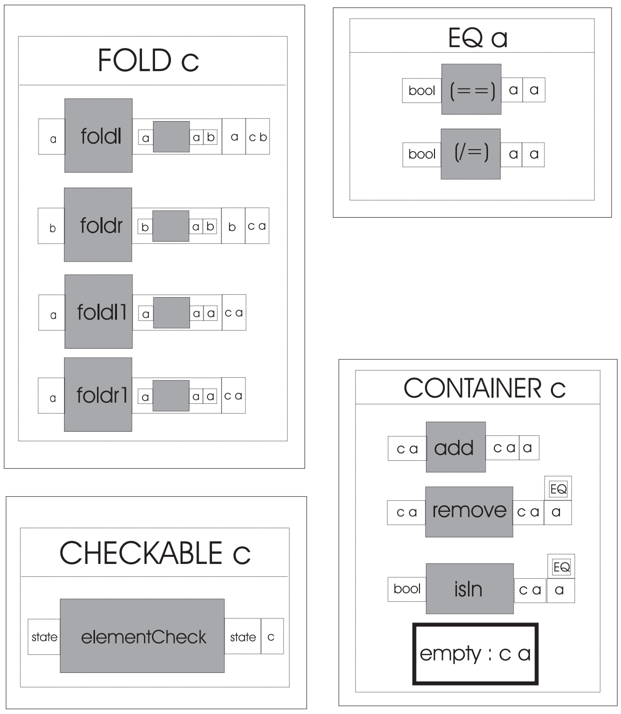

Figure 99: The Permissive Signatures `FOLD`, `EQ`, `CHECKABLE` and `CONTAINER`

Permissive Signature Description Document CASE

Name:	`CHECKABLE`
Version:	20000712:0
Module:	
Parameter(s):	c : *
Operations: *(with type specs.)*	`elementCheck : state -> c -> state`
Inherited Signature(s):	

Description:

This signature specifies the function `elementCheck` that delivers consistency checking over an instantiating type.

Figure 100: Permissive Signature Description Document for `CHECKABLE`

of the failure are added to the state. Conversely if all element checks succeed then the model check succeeds. However, the user may still be informed of potential recursion or breaking of abstraction by including this information in the state.

The ordering of the checking of elements is important. The general approach is that an element should be checked before used. Thus, for example, a type should be checked before one checks its use by a function or another type, and a permissive signature should be checked before checks are applied to its instantiation by a type. We define a partially ordered set $(S, \preceq)$, where S is the set of consistency checks, and for two checks x and y, $x \preceq y$ is defined as *x must be applied in advance of y*. We present a graphical representation of $(S, \preceq)$ in Figure 101. Each check is presented on a node, and for any two checks where x is the immediate predecessor of y, the node x appears above the node y and they are connected by a link. For any two checks s and t where $s \preceq t$, s appears above t and there is a path - or sequence of nodes connected by links - from s to t.

A total ordering which satisfies the partial order is presented in the following enumerated list. In each case we qualify the position of a check in the list by stating those checks that are immediate successor checks.

1. **Uniqueness of type constructors.** Types are fundamental to the development of FAD models and are used in the development of all other micro units. The only check that is required on a type is that it does not reuse a type constructor name. That is, one wants to prevent the use of the same constructor with different kinds. Thus if the type constructor `aType` is currently used with kind `*` and then is reused with kind `* -> *` then this second occurrence is an inconsistency. Multiple use of a type constructor name with the same kind refers to the same type. Hence if one has multiple type dependency diagrams for a single type the conjunction of diagrams must be used. This check must be applied in advance of:

    - uniqueness of micro unit host checks since the type's existence must be checked before it is assigned to a module;
    - uniqueness of permissive signature specifications some of which may use existing types; and,
    - uniqueness of exclusive signature names and micro unit existence checks of

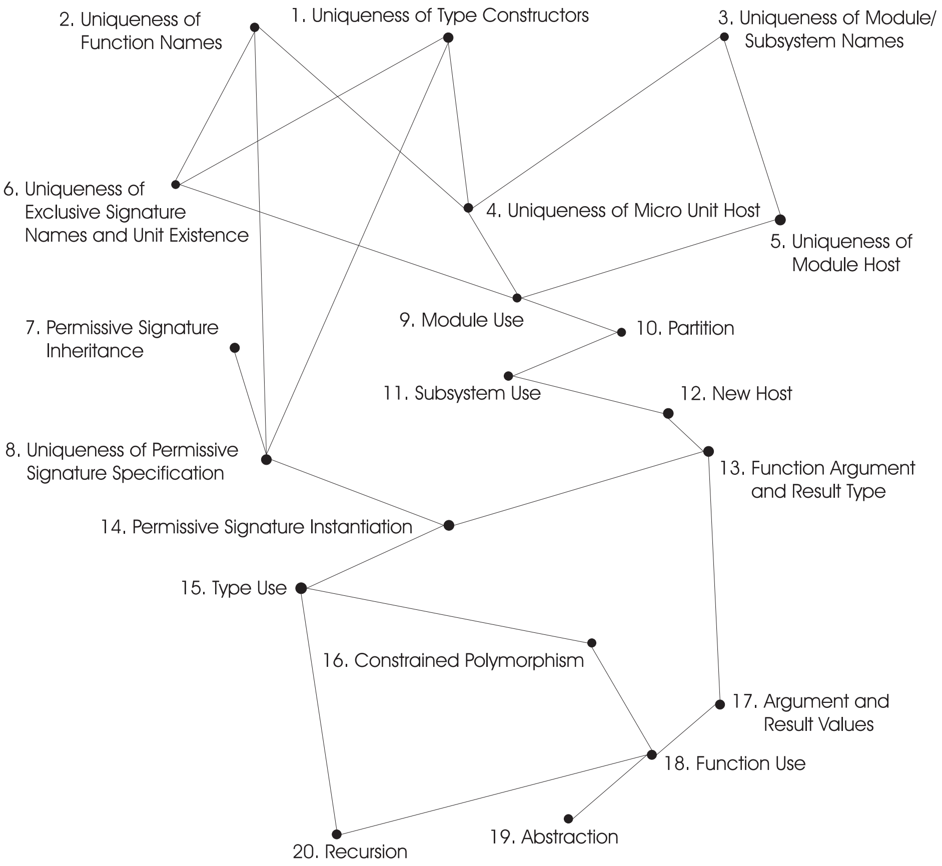

Figure 101: Partial Order for Consistency Checks

which one or more may be a type.

2. **Uniqueness of function names.** Functions are also fundamental to the development of FAD models. As stated in the requirements analysis this check uses a pass/fail check of the arity of functions that share a name, and if this check is passed, a warning check is applied to indicate that the name is being shared. In common with types, multiple use of a function name (and associated types) results in the function adopting the aggregate of the information. This check must be applied in advance of the same checks as check 1 and for equivalent reasons.

3. **Uniqueness of module and subsystem names.** This is simply to prevent one name being used for a module and a subsystem. Modules which share the same name are assumed to be identical and therefore host the aggregate of elements hosted by each. The same rule applies for subsystems. These must be checked in advance of these macro units being used in hosting relationships. That is, they must be applied in advance of uniqueness of micro unit host checks and uniqueness of module host checks.

4. **Uniqueness of micro unit host.** Every micro unit must be hosted by at most one module. These checks must be applied in advance of module use checks which depend on the assignment of micro units to host modules.

5. **Uniqueness of module host**. Every module must be hosted by at most one subsystem. These checks must be applied in advance of module use checks since a module may only use another module that is either hosted by the same subsystem or if either is unhosted.

6. **Uniqueness of exclusive signature names and unit existence**. Each exclusive signature specifies a particular set of micro units. Each of these units must exist in the state before being specified in an exclusive signature. Exclusive signatures whose specifications do not match must have different names. These checks must be applied in advance of module use checks in which exclusive signatures have a mediating rôle.

7. **Permissive signature inheritance**. One way of creating new permissive signatures is by inheriting from and possibly adding to existing signatures. The checks

are based on matching the kinds of the parameters of the inheriting and inherited signatures. That is, for each parameter of the inherited signature(s) there must be a parameter of the same kind in the inheriting signature. These checks must be applied in advance of uniqueness of permissive signature specification checks that may depend on the inheritance association between permissive signatures.

8. **Uniqueness of permissive signature specifications**. This includes checking that any specification appears in at most one permissive signature up to inheritance. Permissive signatures whose specifications do not match must have different names. In addition, one checks that any type used in a specification exists in the state. These must be checked in advance of the use of a permissive signature in a type/permissive signature instantiation.

9. **Module use**. These checks need to be applied in a particular order. The module use relationships in a module dependency diagram should be checked in the following order where we are assuming no recursion in the diagram:

   (a) those at the base of the diagram should be checked first. That is, those for which any item specified in the exclusive signature must be hosted by the associated module should be checked first;

   (b) those in the next layer up should be checked next;

   (c) continue until one reaches the relationship(s) at the top of the diagram which should be checked last.

   Where one has recursive dependencies the use relationships involved in the recursion are checked in any order within the appropriate position in the above ordering.

   Module use checks include:

   - uniqueness of module use relationship checks; and,
   - the exclusive signature mediation 1 checks described in Section A.2.

   They must be applied in advance of partition checks whose success may depend on the relationship between modules in a subsystem.

10. **Partition**. Any partition relationship between a subsystem and a module must be unique. That is, there must be a unique exclusive signature that mediates the relationship. In addition, the exclusive signature mediation 1 checks must be applied to the exclusive signatures and their associated modules. Partition checks must be applied in advance of subsystem use checks that may rely on the partition relationships between server subsystems and their modules.

11. **Subsystem use**. These checks use two checks:

    - uniqueness of the use relationship between any two subsystems. That is, mediation must be through a unique exclusive signature; and,
    - exclusive signature mediation 2 checks, which were described in Section A.2.

    These checks must be applied in an order that takes into account the subsystem interdependencies. They must be applied in advance of new host checks, since a function and one or more of its types may be hosted in modules that are hosted by different subsystems.

12. **New host**. The introduction of hosts for micro units or modules that have been declared in existing models may affect the consistency of elements that have previously passed a check. For example, a function use relationship or type use relationship may now be inconsistent due to changes of the modules which host the related functions. We therefore need to check all elements which are affected by the new host relationships. These checks must be applied in advance of function argument and result type checks whose success may depend on the hosting relationships of types that appear in existing models.

13. **Function argument and result type checks.** The argument and result types of a function must be visible from the function. These checks must be applied in advance of:

    - argument and result value checks since a function can only use values once the visibility of a type has been checked; and,
    - permissive signature instantiation checks that require the existence of the specified functions.

14. **Permissive signature instantiation**. These were described in Section A.2 and must be completed in advance of type use checks that may require used type(s) to instantiate one or more permissive signatures.

15. **Type use**. These were described in Section A.2 and must be applied in advance of:

    - constrained polymorphism checks that may require type/permissive signature instantiation checks between a permissive signature and a type used by an argument or result type; and,
    - recursion checks over types that depend directly on the type use relationships.

16. **Constrained polymorphism**. These were described in Section A.2 and must be applied in advance of function use checks. A function may use a function that requires a type/permissive signature instantiation.

17. **Argument and result values**. These checks were described in Section A.2. A function which is either (partially) applied to its arguments or has a given return value needs to be checked in advance of the use of the function in a function use relationship.

18. **Function use**. These were described in Section A.2 and must be applied in advance of abstraction and recursion checks since they both directly depend on the set of function use relationships.

19. **Abstraction**. These checks were described in Section A.2. These warning checks have no checks which are dependent on their outcome.

20. **Recursion**. These checks were described in Section A.2. These warning checks have no checks which are dependent on their outcome.

Module use (9) and subsystem use (11) checks aside, there is no required ordering of checks of the same sort.

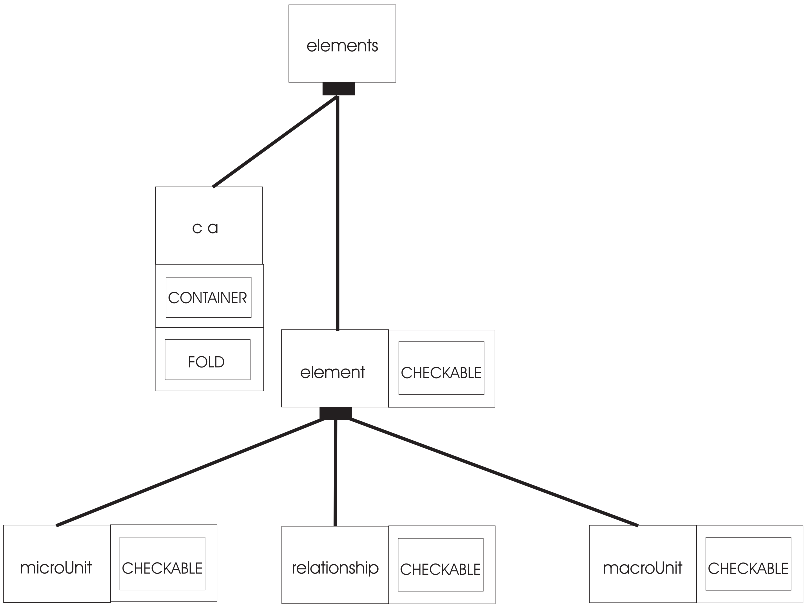

Figure 102: Analysis of type `element`

FAD Description

We now refer to Figure 103. `newModelCheck` uses two functions `checkElements` and `updateState`. `checkElements` takes arguments of type `state` and `elements` and returns a value of type `state`. The `elements` value comes from the `model` value to which `newModelCheck` is applied. `checkElements`'s second argument of type `elements` uses a collection type that is associated with the permissive signature `FOLD` since the function `elementCheck` is folded over the elements. The permissive signature `CHECKABLE` guarantees the existence of the `elementCheck` function.

The type `element` is a union of types that represent micro units, `microUnit`, macro units, `macroUnit`, and relationships `relationship`. Each of these types also instantiate the permissive signature `CHECKABLE` and are themselves unions of types, which we will return to later in the analysis. We present the current design of the type `element` in Figure 102.

The function `checkElements` uses two functions. `applyOrdering` acts as a controller of the application of the element consistency checks. That is, it checks for the existence of different types of elements and applies the relevant `elementCheck` to them using the total order described earlier in this section. `applyOrdering` takes three arguments of type `state`, `elements` and the functional type `state -> element -> state`. The function is partially applied to the value `elementCheck`. `warningChecks` uses the functions `recursionCheck` and `abstractionCheck`, which are non-element specific warning checks. That is, they are not directly bound to a particular element, and their application does not affect the various substate values. `warningChecks` takes a single argument of type `state` and returns a value of the same type. `recursionCheck` and `abstractionCheck` have the same type as `warningChecks`.

The function `applyOrdering` uses two functions. `orderModuleUseChecks` manages the application of the function `moduleUseCheck` and `orderSubsystemUseChecks` provides a similar service for the function `subsystemUseCheck`. That is, they make sure that these particular checks are applied in the appropriate order. Each function takes the collection of the relevant use checks as one of the arguments, and requires the use relationship type to instantiate the permissive signature `ORD` that guarantees an ordering of values of the instantiating type.

Upon completion of the checks the state will require updating. This is implemented by the function `updateState` that manages the state at the termination of a successful or failed check. It simply takes the current `state` value as its argument, since it includes all the information required, and returns the updated `state` value. If the model check was successful then the `subState1` value should be updated to reflect the 'addition' of the elements of the `subState3` value. Addition could mean either the introduction of new elements or the confirmation of the use of existing elements in the new model. In addition, the type `passOrFalse`'s value will indicate success and include a message that reflects this outcome.

If the check of the model failed then the `empty` value of the type `subState3` is returned, and the `passOrFail` value signals failure with a message which describes the details of the failure.

Each type that represents a micro unit, macro unit or relationship - such as the type `function` - uses the type `modelIDs` to record the models in which an element appears. The function `add` defined over the type `elements` uses the function `add` defined over the type `modelIDs` to deliver the required functionality. Both of these functions use the `add` functions guaranteed by the instantiation of the permissive signature `CONTAINER` by the collection types used by the types `elements` and `modelIDs`. In Section A.6 we describe the design of the permissive signature `CONTAINERPLUS` that inherits the functionality specified in `CONTAINER` but enables behaviour that is dependent on the contained item's type.

In the following section we present the scenario analysis when a model is an update of an existing model.

A.3.4 Checking an Update

Informal Description

Checking an update of an existing model requires modification of the state in advance of any consistency check. This is because the state, among other things, is meant to represent the current set of elements against which the check of the model is being applied. Those elements that exist only in the previous version of the model being updated should not influence the checker. Thus the modification of the state involves removing those elements that appear only in the previous version of the model. Obviously if they appear in other models or are repeated in the updated version then they should remain as data in the consistency check.

Once the state has been modified one can proceed with the consistency checks. They must now include not only the checking of elements in the model but also checking for any inconsistencies that may have arisen due to the changes. That is, some elements in the state will need to be rechecked. The model is checked in advance of the modified state. This is because, if one adopts the opposite approach, one may uncover inconsistencies that are due to the non-existence of elements declared in the model. For example, an existing type use relationship may be made inconsistent due to the removal of a module use relationship. However, the updated version of the model may include a new module architecture that satisfies the visibility requirements of the type use relationship.

If the model is checked successfully then one can check for inconsistencies in the state. That is, have any inconsistencies arisen due to the removal of elements from the state? We use the partial order presented in Figure 101 to determine which elements may affect the consistency of existing elements if they are removed from the state. We present each element with the elements they may affect.

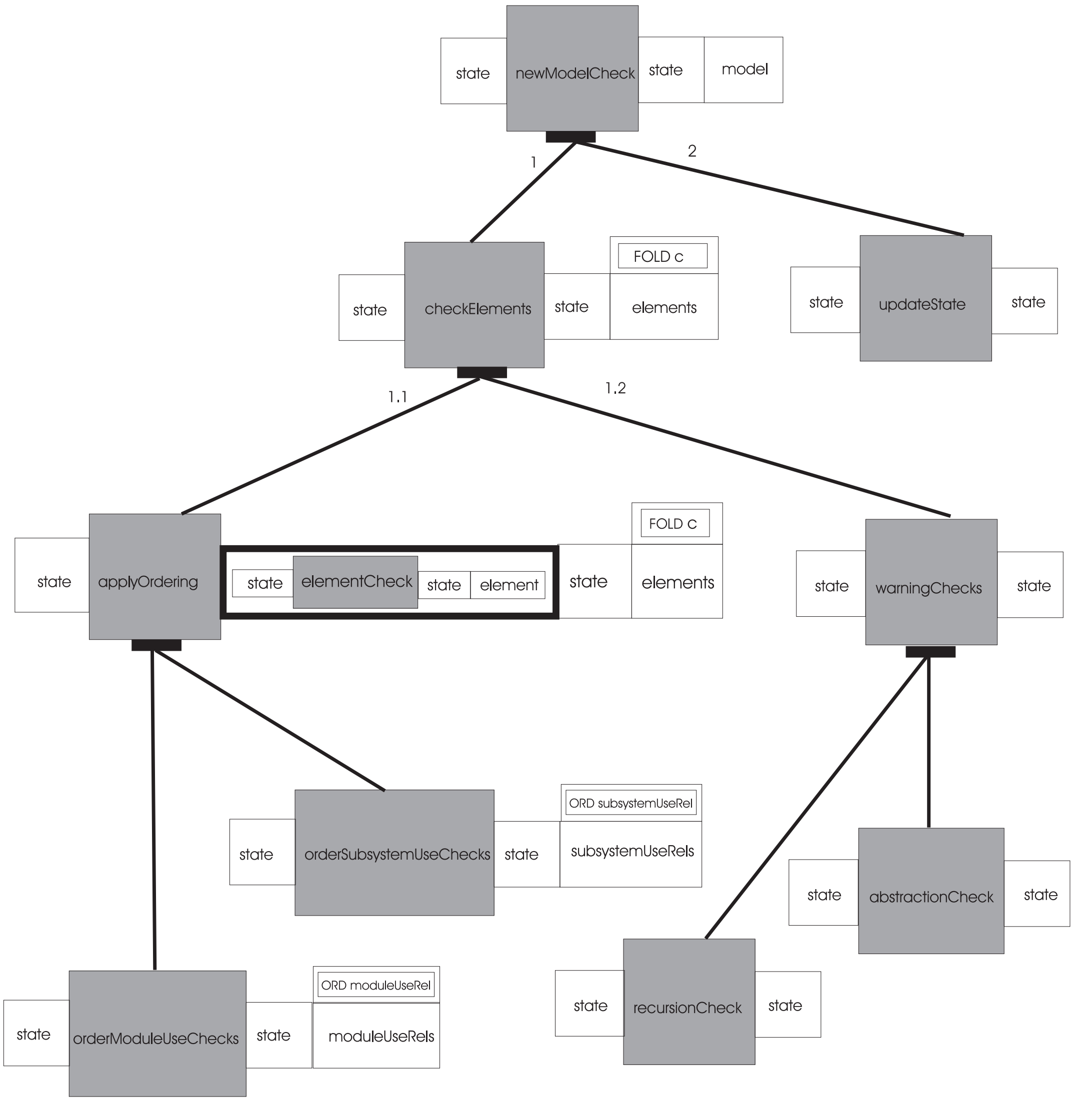

Figure 103: `newModelCheck` Function

Function: the removal of a function may affect:

- the instantiation of a permissive signature since one has to check for the existence of a function of the required type; and,
- the unit existence checks for an exclusive signature since again one checks for the existence of the units specified in the signature.

Type: the removal of a type may affect:

- the specification of a permissive signature that may include one or more units whose types include the removed type; and,
- the unit existence checks for an exclusive signature since one checks for the existence of the units specified in the signature.

Permissive Signature: the removal of a permissive signature will have no effect. If it doesn't exist in any models then it is not being used either in association with a type or in the construction of a new signature through the inheritance relationship.

Module: the removal of a module will have no detrimental effect on the consistency of existing elements since any units that were previously assigned to the module will now be visible from any client unit;

Subsystem: the same result as for modules.

Module Use Relationship: the removal of a module use relationship may result in previously visible units becoming invisible to their clients. This may affect the consistency of type use, function use, function argument and result type relationships and other module use relationships.

Partition Relationship: the removal of a partition relationship may result in previously visible units becoming invisible to their clients. This may affect the consistency of type use, function use, function argument and result type relationships.

Subsystem Use Relationship: the removal of a subsystem use relationship may result in previously visible units becoming invisible to their clients. This may affect the consistency of type use, function use, function argument and result type relationships and other subsystem use relationships;

Exclusive Signature Specification: the removal of a micro unit from an exclusive signature may also affect any relationship that depends on the visibility of that unit.

Type/Permissive Signature Instantiation: the removal of a type/permissive signature instantiation may affect the consistency of functions with types associated to permissive signatures. That is, a constrained polymorphism check depends on the existence of the required type/permissive signature instantiations.

Type Use Relationship: the removal of a type use relationship may affect the consistency of functions with types associated to permissive signatures. That is, a constrained polymorphism check may depend on a type use relationship between an argument or result type and the type that instantiates the permissive signature.

Those elements that exist in the previous version of the model but are absent from the new version therefore provide a guide for the checks that are required on the remaining state. One should not simply recheck all elements of the types indicated above, but rather those elements that have an association with the removed element. For example, if a type use relationship is removed, then one has knowledge of the client and server types and this should guide the constrained polymorphism checks that need re-application.

FAD Description

We now refer to the update of the function dependency diagram for `modelCheck` presented in Figure 104. The original diagram was presented in Figure 95. The function `modifyState` modifies the state. `modifyState` uses `singleUse`, which takes arguments of type `state` and `model` and returns a value of type `elements`, which represents those elements that only appear in the previous version of the model being updated. `singleUse` makes use of the `modelIDs` value that is used by the types that represent each form of FAD element. For example, the micro unit types `type`, `function` and `permSig` each use the type `modelIDs`. This design is presented in the model in Figure 108 at the end of this section, and `singleUse` is further analysed in Section A.5.4.

Each element returned by `singleUse` is removed from the `subState1` value using `remove` and added to the `subState3` value using `add`. The `subState3` value is initially

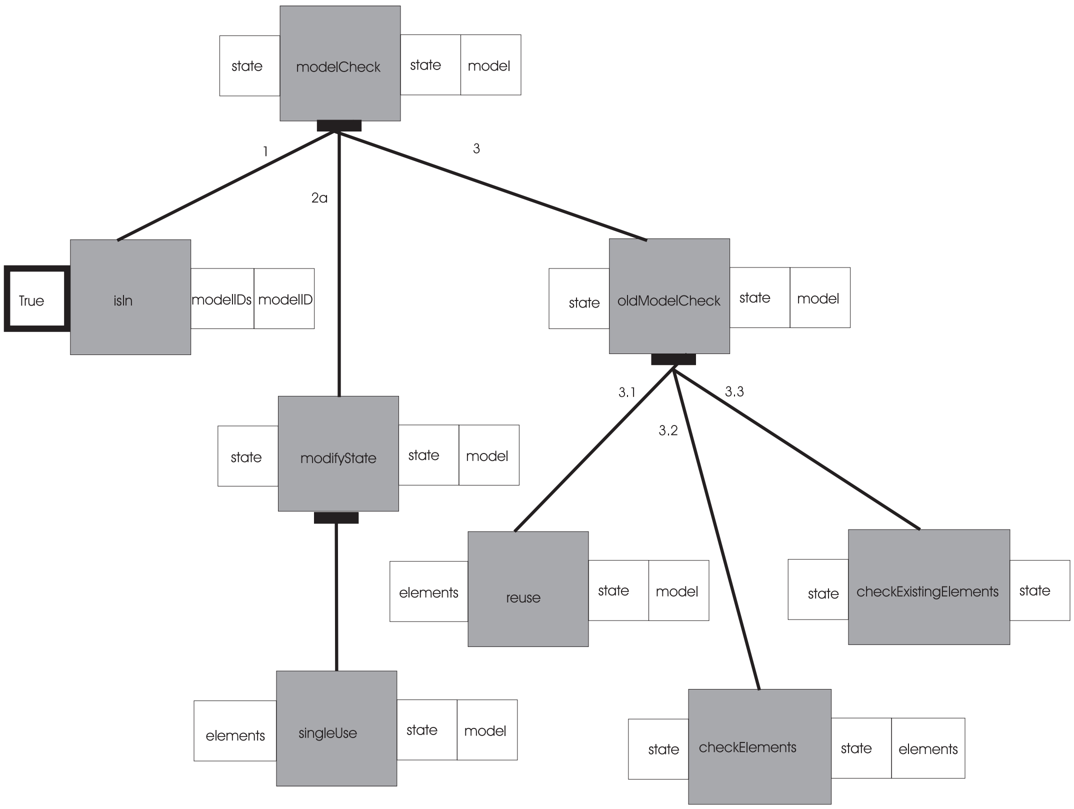

Figure 104: Update of `modelCheck` Function

empty. If the model check fails then one can recreate the version of the state prior to checking by returning the empty `subState3` value and adding the previous `subState3` value to `subState1`. The functions `add` and `remove` are guaranteed by the permissive signature `CONTAINER` instantiated by the collection type used by the type `elements`. If an element appears in other models it remains in the `subState1` value but the `modelID` value of the model being checked is removed from its `modelIDs` value.

The modified state provides the first argument for the function `oldModelCheck`. This function uses three functions that are applied in the order of the following list:

- the function `reuse` is called and returns those elements that are used in both the previous version and updated version of the model. These elements do not need to be rechecked;

- the function `checkElements` is applied to the current `state` value, and those elements of the model not returned by `reuse`. That is, those elements that are new to the model; and,

- if the previous check terminates successfully then the state is checked for inconsistencies using the function `checkExistingElements` that takes a single argument of type `state`. This function manages the consistency checks applied to elements that existed prior to the check of the current model. For example, any type use relationship whose host modules are no longer associated through a module use relationship needs to be rechecked for consistency. The removal of the relationship does not by default imply an inconsistency since another module use route may exist. Thus `checkExistingElements` uses the `subState3` value to determine those checks that are required of the elements of the `subState1` value. The function uses the `elementCheck` function to apply the relevant checks.

We have concentrated thus far on the checking of models. In the following section we present some illustrative examples of analyses of element check functions upon which the model checks largely depend.

A.4 A Selection of Element Check Analyses

In this section we present a representative sample of analyses of element check functions. Those selected highlight both the similarities in their behavioural requirements and cover the interaction between units of the same sort and those of different sorts. We therefore present a micro unit use check, a macro unit use check and a non-use check, which is an example of what is required when checking the interaction between units of different sorts. The first analysis that we present is that of the function which checks function use relationships.

A.4.1 Analysis of `functionUseCheck`

Informal Description

If the element - a `functionUseRel` value - is present in the state then there is no need to further check its consistency since its presence implies that it has previously satisfied a check. However, one needs to update the element entry in the state to include its appearance in the model being checked. This is true of all element checks. Thus the first requirement of any check is to determine whether the element is present in the state. If the function use relationship does not exist then the use of one function by another function is consistent if and only if the used function is *visible from* the using function.

FAD Description

We now refer to Figure 105. The function `functionUseCheck` takes two arguments of type `state` and `functionUseRel` (the type of function use relationships) and returns a value of type `state` that will reflect the outcome of the check. `functionUseRel` first tests for the existence of the relationship using the function `inState`. `inState` takes two arguments of type `state` and `functionUseRel` (which is required to instantiate the permissive signature `EQ`) and returns a Boolean value that indicates whether the item exists in the state or not. The permissive signature instantiation is required since one wants to match the relationship against one in the state.

If the relationship exists in the state then the check is terminated, and the `state` value is updated to record success and the fact that the element appears in the model. If it does not exist, `functionUseCheck` uses the function `visibleFrom` to test whether the used function is visible from the using function. The function `visibleFrom` takes an argument of type `state` and two of type `function` and returns a `bool` value. It requires the state argument since the various hosting and use relationships are stored in the state.

If the application of the function `visibleFrom` returns `True` then one adds the function use relationship to the state using `addToState`. This function uses the `add` function where the first argument is of type `elements` (whose value comes from the `subState1` value). If the visibility check fails then `functionUseCheck` calls the function `reportFailure` that returns the state, where the value of type `passOrFail` includes a message indicating the failure.

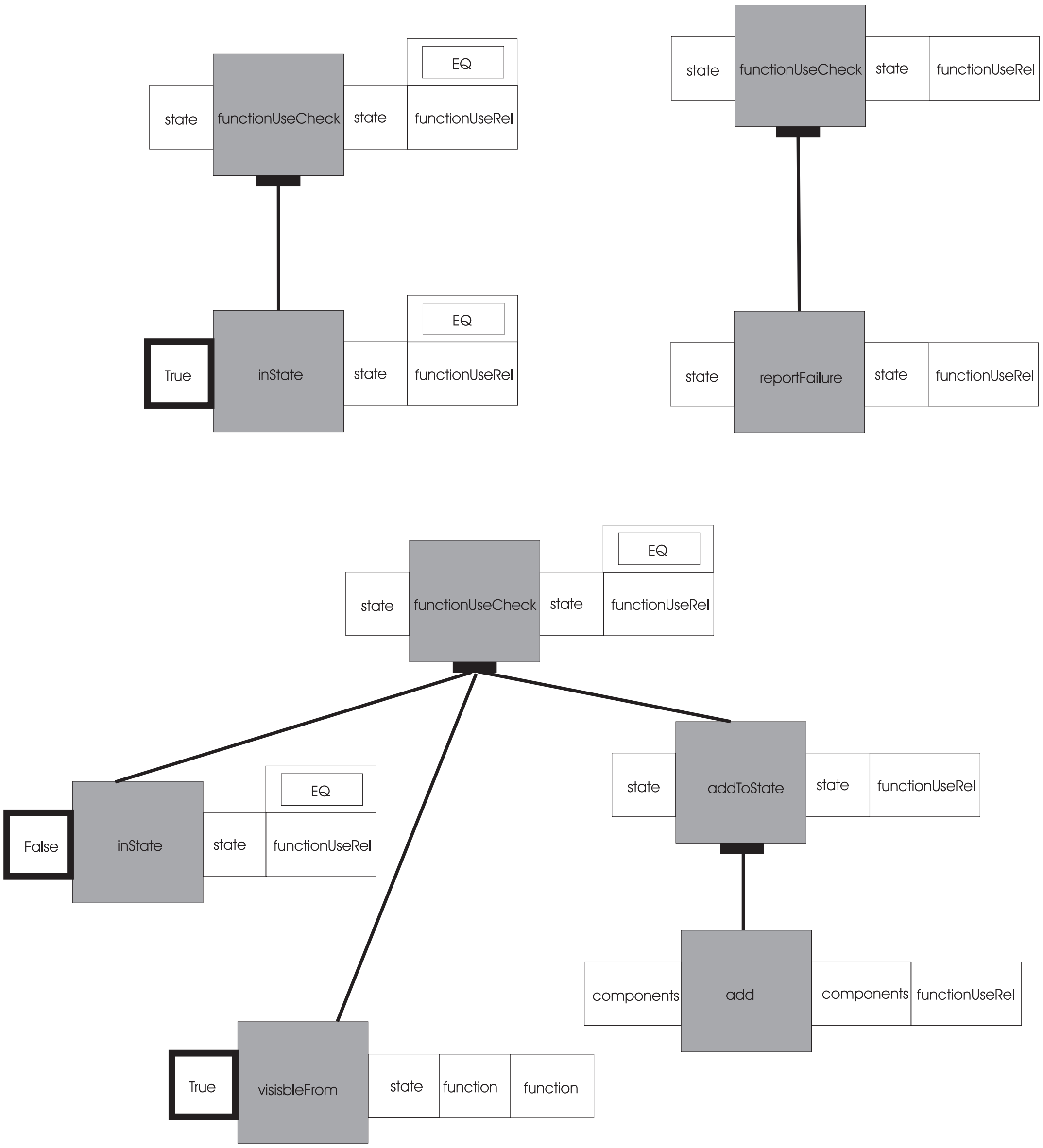

Figure 105: Analysis of `functionUseCheck`

A.4.2 Analysis of `moduleUseCheck`

Informal Description

A module use check begins in the same manner as the previous check. That is, one tests for the existence of the relationship in the state. If it exists then one terminates the check successfully. The module use relationship of the model is the same as one in the state if the client and server modules are the same in each case, and the exclusive signature that mediates the relationship matches. If it does not exist then one has to test for the uniqueness of the relationship between the stated modules. That is, any two modules **M** and **N** should have at most one module use relationship where **M** is the client. This means that the use of module **N** by module **M** should be mediated by a unique exclusive signature. If the test fails then the check is terminated and the user informed of the problem. If the test succeeds one checks that the server module is visible from the client module.

Once again if this test fails the check is terminated and the user informed of the failure. If the check succeeds, the association between the mediating exclusive signature and the server module needs to be checked. If this check succeeds then the whole check is successful and the state can be modified to reflect this.

FAD Description

We refer now to the function dependency diagram of Figure 106 and the function description document in Figure 107. The function `moduleUseCheck` takes two arguments of type `state` and `moduleUseRel` (the type of module use relationships) and returns a value of type `state`. It uses the functions:

- `inState` that takes the same argument types as `moduleUseCheck` but returns a value of type `bool`, which reflects whether the element exists in the state or not;
- `unique` that tests for the uniqueness of the relationship and has the same type as `inState`. In an optimised implementation one may merge `inState` and `unique` into a single function that returns a pair of Boolean values;
- `visibleFrom` that takes an argument of type `state` and two of type `module` and returns a value of type `bool`; and,

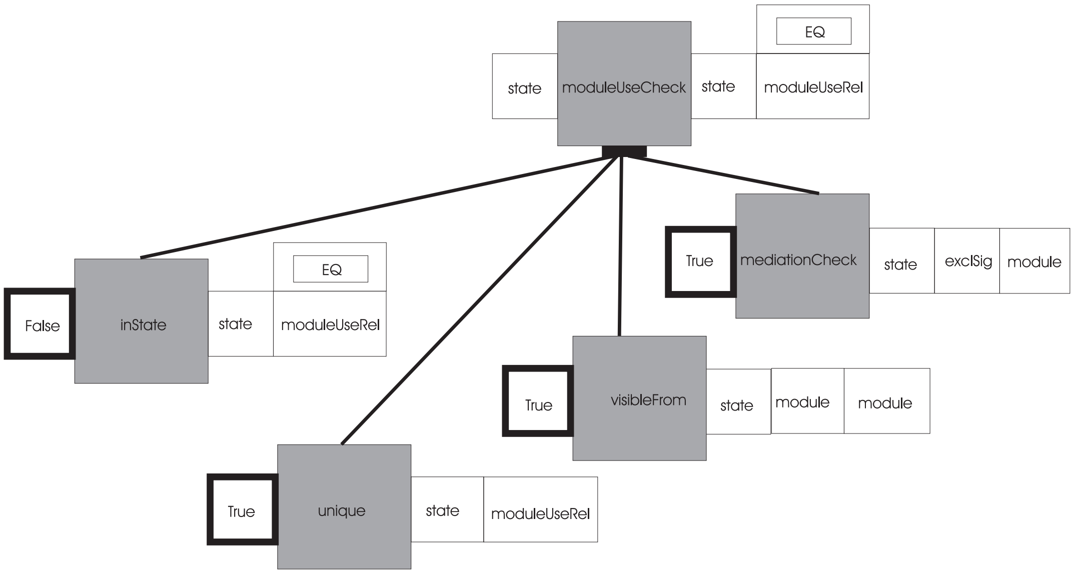

Figure 106: Analysis of `moduleUseCheck`

- `mediationCheck` that checks the association between the exclusive signature and module. It takes three arguments of type `state`, `exclSig` and `module` respectively. The `state` value contains the existing elements which may be called upon to confirm the consistency of the association between the exclusive signature and the module. This function implements the exclusive signature mediation 1 check described in Section A.2.

If `inState` returns `True` then the check terminates successfully. In the case that `inState` returns `False`, if any of the other functions returns `False` then the check terminates unsuccessfully.

A.4.3 Analysis of `typePermSigCheck`

Informal Description

The association between a permissive signature and one or more types (the number depends on the number of parameters of the permissive signature) initially proceeds in a similar fashion to the previous checks. That is, one checks for the existence of the relationship in the state. If it exists the check terminates successfully. If it doesn't then one needs to check that the permissive signature is visible from the type(s). A sensible

Function Description Document CASE

Name:	`moduleUseCheck`
Version:	20000712:0
Module:	
Arity:	2
Type Specification:	`state -> moduleUseRel -> state`
Contract Association:	
Instantiations:	`EQ moduleUseRel`

Functions Used:	`inState : state -> moduleUseRel -> bool`
	`unique : state -> moduleUseRel -> bool`
	`visibleFrom : state -> module -> module -> bool`
	`mediationCheck : state -> exclSig -> module -> bool`

Description:

If any element exists in the state then there is no need to further check its consistency since its existence implies that it has previously satisfied a check. This is true of all element checks. Thus the first requirement of any check is to determine whether the element exists in the state.

The module use relationship of the model matches one in the state if the client and server modules match in each case, and the exclusive signature which mediates the relationship matches. If it does not exist then one has to test for the uniqueness of the relationship between the stated modules. That is, any two modules **M** and **N** should have at most one module use relationship where **M** is the client. This means that the use of module **N** by module **M** should be mediated by an unique exclusive signature. If the test fails then the check is terminated and the user informed of the problem. If the test succeeds one checks that the server module is visible from the client module.

Once again if this test fails the check is terminated and the user informed of the failure. If the check succeeds the association between the mediating exclusive signature and the server module needs to be checked. If this check succeeds the the whole check is successful and the state can be modified to reflect this.

Figure 107: Function Description Document for the Function `moduleUseCheck`

design is one where all permissive signatures are visible from all types. Thus if this check fails this should provide a warning signal regarding the design.

Upon successful completion of the visibility check one checks that the permissive signature and type(s) have matching kinds. That is, the kind required by each parameter of the permissive signature is matched by the kind of the type constructors of the instantiating types. If this check succeeds then one needs to check for the existence of the units specified in the permissive signature.

FAD Description

The function `typePermSigCheck` takes two arguments of type `typePermSigRel` (the type of type/permissive signature relationships) and `state` and returns a value of type `state`. It uses the functions:

- `inState` that takes the same argument types as `typePermSigCheck` but returns a value of type `bool`;
- `visibleFrom` that takes three arguments of type `state`, `permSig` and `type` and returns a value of type `bool`. This is the third occasion that we have used a function called `visibleFrom` and in each case with a different type. In Section A.6 we use this as an illustrative case of function development guided by name reuse;
- `kindCheck` that takes two arguments of type `permSig` and `type` and returns a `bool`; and,
- `allInState` that takes the same arguments as `visibleFrom` and returns a `bool`. It uses the functions `inState` to determine whether each specified unit of the required type exists in the state. In Section A.6 we take the various `inState` functions and design a single function in their place;

The types `permSig` of permissive signatures and `type` of types must both use the type `kind`, the set of kind values. We present the design of the type `microUnit` in Figure 108 and the type description document for the type `permSig` in Figure 109.

That completes our selection of element checks. In the following section we describe the development of an initial module architecture for the subsystem.

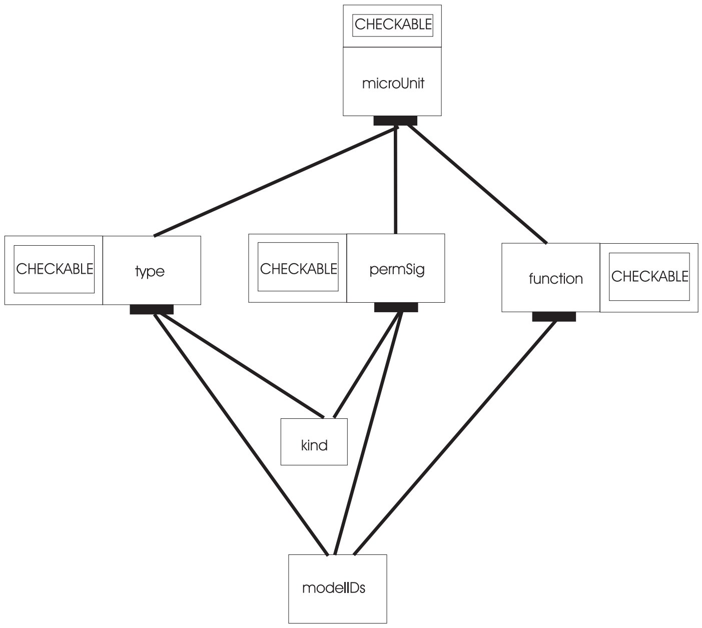

Figure 108: `microUnit` Type Design

Type Description Document CASE

Constructor Name: `permSig`
Version: 20000713:0
Kind: `*`
Module:
Types Used: `kind`, `modelIDs`
Parameters:
Permissive sigs.: `CHECKABLE`
Description:
The type `permSig` is the type of permissive signatures. Each signature has a kind and a record of the models in which it is used. As with all elements a permissive signature must support consistency checking.

Figure 109: Type Description Document for the Type `permSig`

A.5 Module Architecture

In this section we introduce a module architecture into the system. This involves the declaration of modules, the assigning of micro units to host modules, and the introduction of module use relationships between modules and partition relationships between the subsystem and some of its modules. These relationships require the development of exclusive signatures to mediate access to hosted units.

The guiding principle here is to host a type with the functions that deliver the required behaviour over the type, and to put an abstraction barrier around the type. That is, each type should be hosted in its own module with the operations over the type. When it is specified in an exclusive signature it should be specified, if possible, without its constructor signature. The cost of this approach is that one may require *get* and *set* functions to support access to the type by clients that are external to the abstraction barrier. Although this is a sensible way initially to develop a module architecture (and moreover one that will enable the localization of future changes), it is unlikely that it will result in a design devoid of imperfections. There are occasions where one may need to introduce modules that do not host any types but:

- manage the interaction between two or more types hosted elsewhere;
- present a collection of polymorphic functions that are linked by the behaviour that they implement;
- present a collection of constrained polymorphic functions that are linked by the permissive signature(s) that need to be instantiated; or
- simply avoid overburdening a module with an excessive number of units such that it becomes difficult to manage.

The initial foci of attention are therefore the types. Once decisions have been made regarding the required modules for hosting the types one assigns the remaining micro units to the appropriate modules. Scenario and type dependency analyses are then applied where extra information is required due to the design of the module architecture.

In the following section we develop an initial module architecture for the subsystem using *module architecture analysis.* Once again we will first present a textual description of the analysis followed by a description of the development of FAD models.

A.5.1 Module Architecture Analysis

Informal Description

Each of the types which are exported from `ConsistencyCheckerSS` is assigned to its own module. One then assigns those functions that implement the behaviour required over a type to the same module as the type. If a function implements behaviour over more than one type, one assigns it to a module that hosts one of the types, develops the module architecture with the required module dependencies, and then analyses the design of the architecture. For example, one may require mutual dependency between modules or one may discover that a collection of functions are best hosted by a module that delivers a particular functionality that may be reusable over more than one type or collection of types. That is, at this stage of development one is trying to minimize the number of modules and to emphasize the need to localize functions and their types.

The subsystem supports the consistency checker of the CASE tool and exports model checking functionality as well as the types directly associated with model checking. The details of the implementation of model checking are of no interest to clients within the system. That is they will be presented with a minimal interface to the types and function(s) associated with model checking. This enables both incremental development of parts of the system, and minimal disruption due to maintenance or extension of the system.

FAD Description

We now refer to Figure 110. The subsystem `ConsistencyCheckerSS` is associated with the exclusive signature `CCSIG` that mediates access to the subsystem. The function `modelCheck`, which implements the consistency checking of a model, and the types `state` and `model` are specified in the signature. Thus the types `state` and `model` are our initial foci. They each need to be assigned to a module to which access is controlled by an exclusive signature. Each exclusive signature will initially specify the type(s) they are hosting with any required functionality added during development. The modules are `StateMod` and `ModelMod` respectively. Each of these modules will be associated to `ConsistencyCheckerSS` through a partition relationship that is mediated by an exclusive signature. Other modules of the subsystem have no partition relationship with the subsystem and are not specified in the mediating exclusive signatures, since their units should remain invisible to any clients of the subsystem.

We now proceed with type and function host analysis applied to the types used by `state` and `model`, and the functions used by `modelCheck`. With reference to Figures 98 and 110, the five types used by the type `state` are assigned to different modules:

- `subState1` is hosted by `SubState1Mod`;
- `subState2` is hosted by `SubState2Mod`;
- `subState3` is hosted by `SubState3Mod`;
- `modelIDs` is hosted by `ModelIDsMod`; and,
- `passOrFail` is hosted by `PassOrFailMod`.

Similarly we assign the two types used by the type `model` to two separate modules:

- `elements` is hosted by `ElementsMod`; and
- `modelID` is hosted by `ModelIDMod`.

Immediately one can sketch an initial module architecture that satisfies the visibility requirements of the types `state` and `model`. For example, the module `StateMod` uses the modules that host the types used in its construction. That is, `SubState1Mod`, `SubState2Mod`, `SubState3Mod`, `ModelIDsMod` and `PassOrFailMod`. Both `ModelMod` and `ModelIDsMod` use `ModelIDMod`, and `ElementsMod` is used by `ModelMod`, `SubState1Mod`, `SubState2Mod` and `SubState3Mod`.

We now proceed with *function host analysis* in which we assign functions to their relevant host module.

A.5.2 Function Host Analysis

Informal Description

Each of the functions that appears in function dependency diagrams is assigned to a host module. We use the modules described in the previous section as the hosts. If none of these modules is appropriate then either a new module is introduced or the function should be the responsibility of a different subsystem.

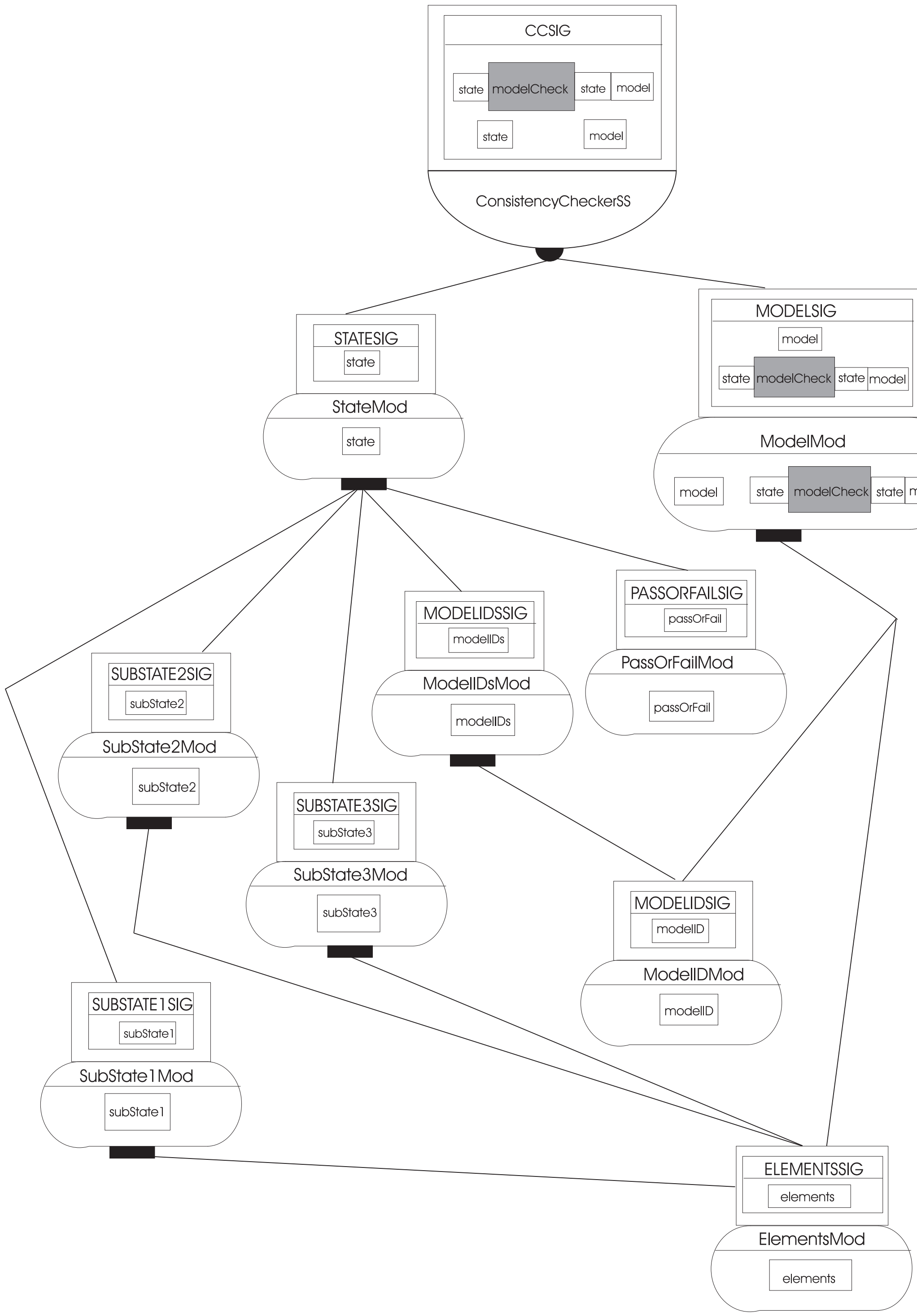

Figure 110: Initial Design of Subsystem `ConsistencyCheckerSS`

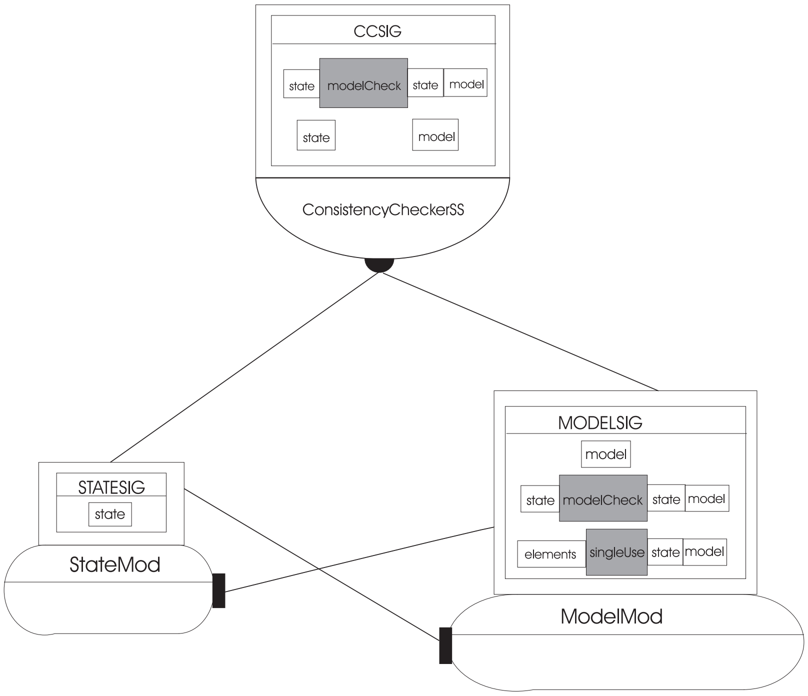

Figure 111: Update of Module Architecture

FAD Description

We present a summary of the outcomes in Tables 8 and 9. Each function is listed alongside its host module with some commentary supporting the assignment. This commentary includes any module use relationships that are required. In Figure 111 we present the new relationships between the modules `StateMod` and `ModelMod`, which also uses the results of the exclusive signature development described in Section A.5.3. The recursive dependency between the modules (and the modules `StateMod` and `ElementsMod`) will be highlighted by the *warning check* on recursion and suggests a poor module architecture design. In this instance the recursive dependency is present in a single model. However, the dependencies could have been described in two different models, and where there are intermediate modules, several models may require investigation to unearth the recursion. An alternative design that avoids recursion is presented in Section A.6.

Initially exclusive signatures only specify the types that they host, but once functions are assigned to modules their associated signatures must be changed in order to avoid

Function	Module	Comment
`modelCheck`	`ModelMod`	This function takes two arguments of types `state` and `model`. However, it is the `model` type whose behaviour it implements. The module `StateMod` must be used by `ModelMod`.
`isIn`	**No Assignment**	This function is specified in the permissive signature `CONTAINER` and will be hosted with whichever collection type is used by `modelIDs`. This module will be hosted by the subsystem that delivers the general basic types and permissive signatures since it is not specific to consistency checking.
`newModelCheck`	`ModelMod`	This function delivers functionality over the type `model`.
`modifyState`	`StateMod`	This function requires access to the construction of the type `state` and delivers functionality over the type. The module `StateMod` uses the module `ModelMod`.
`oldModelCheck`	`ModelMod`	This function delivers functionality over the type `model`.

Table 8: Function Host Analysis

Function	Module	Comment
`updateState`	`StateMod`	This function requires access to the construction of the type `state` and implements functionality over the type.
`checkElements`	`ElementsMod`	This function implements a behavioural requirement of the `elements` type. The module `ElementsMod` uses the module `StateMod`.
`applyOrdering`	`ElementsMod`	This function requires access to the construction of the type `elements` and implements functionality over the type. The module `ElementsMod` uses the module `ElementMod`.
`reuse`	`ModelMod`	This function tests for the reuse of elements in the update version of a model.
`singleUse`	`ModelMod`	This function requires access to the construction of the type `model` and implements functionality over the type.
`warningCheck`	`StateMod`	This function implements functionality over the type `state`.
`abstractionCheck`	`StateMod`	This function implements functionality over the type `state`.
`recursionCheck`	`StateMod`	This function implements functionality over the type `state`.

Table 9: Function Host Analysis (continued)

inconsistency. It is the user who has to ensure this. This process is described in the following section.

A.5.3 Exclusive Signature Analysis

The development of a system based on abstraction and encapsulation requires a module architecture in which exclusive signatures mediate access to the units hosted by each module. Initially we associate a single exclusive signature with each module. This simplifies the initial development of use relationships by providing the developer with a single interface to any macro unit. In addition, it emphasises the importance of encapsulation early in development by making explicit all that an external client may know. Later in development, multiple exclusive signatures for a single module are designed that deliver the required mediation for a particular relationship and thus make explicit exactly what an external client needs to know. That is, a partition relationship between subsystem **S** and module **M** is likely to require a different exclusive signature to that which mediates the use relationship from module **N** to module **M**.

Units are specified in an exclusive signature if they are either used in a use relationship, or are permissive signatures associated with one or more types. Thus one scans the type dependency and function dependency diagrams for those units which should be specified in an exclusive signature. Constructor signatures will not initially appear in any exclusive signatures.

We will illustrate this analysis with the development of the exclusive signature `MODELSIG` that mediates access to the module `ModelMod`.

Informal Description

`MODELSIG` initially specified the type `model`. However, it hosts some functions that need to be visible to clients that are external to the module. The obvious example is the function that implements model checking, which must be visible to clients outside of this subsystem. Thus it must be specified in the exclusive signature that mediates the partition relationship between the subsystem and the module.

In light of the host assignments described in Tables 8 and 9, the use relationship between `modifyState` and `singleUse` requires `singleUse` to be specified in the exclusive signature that mediates access to `ModelMod`. In Section A.6 we illustrate exclusive signature design with those that mediate access to the module `StateMod`.

All other functions of the module are used only by functions of the same module and therefore are not specified in the exclusive signature.

FAD Description

We refer now to Figure 111. Only three units need to be specified in `MODELSIG`:

- the type `model` that is used, for example, by the function `modifyState` which is hosted by the module `StateMod`;
- the function `modelCheck` that implements the externally visible functionality supported by the module `ModelMod`; and,
- the function `singleUse` that is used by the function `modifyState` that is hosted by the module `StateMod`.

The functions `newModelCheck`, `oldModelCheck` and `reuse` are used by functions of the same module and do not have any clients from other modules or subsystems. They do not therefore need to be visible from external clients and hence are not specified in the exclusive signature. The module `ModelMod` is used by the module `StateMod` since the function `singleUse` must be visible from the function `modifyState`. This results in an update of the module architecture.

The introduction of abstraction barriers incurs a cost on the developer. One has to introduce operations which replace direct access to the construction of a type. Further analyses should be applied to discover such requirements. We present such an analysis in the following subsection.

A.5.4 Scenario Analysis of the Function `singleUse`

In this section we provide an analysis of the function `singleUse` that is influenced by the current module architecture.

Informal Description

`singleUse` returns the elements of a model that only appear in the previous version of the model. Thus one needs access to the elements in the state and those in the current version of the model. Each state element whose model identifiers include the current model identifier are checked against the elements of the model. Any which are not members of the model's elements are returned by the function.

FAD Description

We refer here to the function dependency diagram in Figure 112 and to the *Function Description Document* in Figure 113. `singleUse` requires access to the elements of the model and of the state. Since `singleUse` is hosted by the same module as the type `Model` it doesn't need to call any *get* functions on the type. Thus the elements of the model can be accessed directly, but those of the state require the function `getSubState1` to be called with a `state` argument. This returns the `subState1` value, which provides the argument for `getElements`. The functions `getSubState1` and `getElements` are required since the types `state` and `subState1` are abstract relative to the function `singleUse`.

The function `selectInModel` returns those elements in the state which appear in the previous version of the model. It takes an argument of type `elements` and another of type `modelID`, and returns those elements for which the model identifier is included in the `modelIDs` value. `selectInModel` uses the function `getModelIDs`, which takes an argument of type `element` that is abstract relative to the function `selectInModel` that is hosted with the type `elements`. `getModelIDs` returns the identifiers of the models in which an element appears. The `modelID` argument of `selectInModel` is accessed directly.

The collection type used by `elements` needs to support *filtering* behaviour to implement `selectInModel`. This is guaranteed by the association with the permissive signature `FILTER c` which specifies the function `filter : (a -> bool) -> c a -> c a`.

The elements returned by `selectInModel` are each tested for membership of the new version of the model using the function `setDiff`. This function takes two arguments of type `elements` and returns those elements from the first argument that are not in the second. That is, it returns those elements which are not in the new version of the model. `setDiff` uses the function `isIn` that is guaranteed by the permissive signature `CONTAINER` associated with the collection type used by the type `containers`.

This completes our selection of analyses. In the following section of this appendix

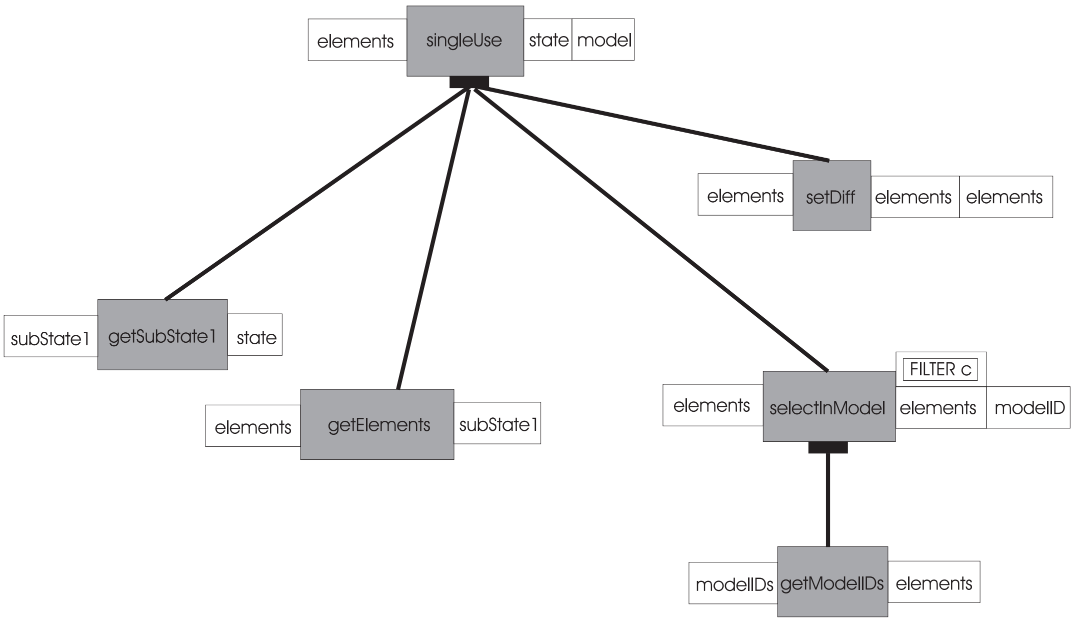

Figure 112: Function Dependency Diagram for `singleUse`

we give some illustrative examples of design phase development.

A.6 Design of `ConsistencyCheckerSS`

Design focuses on the delivery of a solution-domain focused model of the system. That is, where analysis is tied to the problem-domain albeit described in terms of the required paradigm, design aims to produce a system which can be implemented in as an efficient and effective manner as possible. However, the two phases are not mutually exclusive and, for example, modularity, both in macro unit and micro unit development, has had a design impact within the analysis phase of development.

During the design phase, one takes the deliverables of the analysis phase and, using the various mechanisms provided by the paradigm, designs the various micro and macro units such that an efficient implementable design is returned. The transition from a largely analytical model to an implementable design is supported by the consistent paradigm-focus of the methodology and the fact that the diagrams, and many of the techniques used during analysis, are the same as those used during design.

Thus upon completion of this phase one wants:

Function Description Document	CASE
Name:	`singleUse`
Version:	20000720:0
Module:	`ModelMod`
Arity:	2
Type Specification:	`state -> model -> elements`
Contract Association:	
Instantiations:	

Functions Used:	`getElements : subState1 -> elements`
	`getSubState1 : state -> subState1`
	`selectInModel : elements -> model -> elements`
	`setDiff : elements -> elements`

Description:

`singleUse` returns the elements of a model that only appear in the previous version of the model. Thus one needs access to the elements in the state and those in the current version of the model. Each state element whose model identifiers include the current model identifier, are checked against the elements of the model. Any that are not members of the model's elements are returned by the function.

Figure 113: Function Description Document for the Function `singleUse`

- a module architecture built on reusable units with minimal interfaces to other units;
- exclusive signatures that are designed to mediate a specific relationship;
- permissive signatures which are developed to guarantee a particular behaviour and support reuse; and,
- to make use of functional programming's glue such as parametric polymorphism and higher-order functions.

We present in the following subsections some illustrative examples of element design. We begin by updating the module architecture of `ConsistencyCheckerSS`. In Section A.6.2 we present the (related) design of exclusive signatures that mediate access to the module `StateMod`. In Section A.6.3 we describe the development of the permissive signature `CONTAINERPLUS`, and in Section A.6.4 the (related) design of the type `elements`. In Section A.6.5 we describe the development of the functions `visibleFrom`, `visibleFromModule` and `inState`, and finish with a brief summary of the the remaining work to be done.

A.6.1 Module Architecture Design

Informal Description

The current module architecture includes a mutual dependency between the modules `ModelMod` and `StateMod`. This is because they each host the type for which they are named, and each host functions that use the type hosted by the other module. The various *get* and *set* functions must remain in the same module as the type to which they apply because they require direct access to the construction of the type.

However, one may require modules that host functions separately from the types over which they are defined. This is either because the function does not sit naturally with a particular type, or because one requires a module to deliver a particular set of behavioural requirements rather than to host a type and its related functions. For example, the Haskell 98 libraries [101] `List` and `Monad` are in turn, a module that hosts functions over a type hosted by another module, and a module whose functions are defined over a collection of types related by the functionality they support.

Here we introduce the module `CheckMod` that hosts the functions that implement checking functionality but does not host any types. This module hosts the functions that implement the checking functionality required over the types `state` and `model`. That is, the module manages the interaction between these types and therefore uses the modules that host the types. In addition, it provides a single entry route into the module architecture for external clients and a single focus for checking behaviour.

The modules `ElementsMod` and `StateMod` also exhibit a mutual dependency. Once again one can reassign the functions that implement the checking behaviour over these types to a module that uses the above modules. As with `CheckMod` this module manages the interaction between the types hosted by these modules.

FAD Description

We refer now to Figure 114 and to the *Module Description Document* in Figure 115. The module `CheckMod` provides both a single entry point into the module architecture, and collects together the main functions that implement the model checking functionality required by the consistency checker. The exclusive signatures associated with the modules `StateMod` and `ModelMod` now include several *get* and *set* functions that are used by the functions hosted by `CheckMod`.

The module `StateMod` is associated with two exclusive signatures:

- `STATESIG1`, which mediates access to clients in the module `CheckMod`; and,
- `STATESIG2`, which mediates access to clients in the module `ModelMod`.

Thus `updateState`, which is used by `newModelCheck`, is specified in `STATESIG1` but not in `STATESIG2`. The function `reuse` is now specified in the exclusive signature `MODELSIG` since its client function `oldModelCheck` is now hosted in a different module.

We present a similar design in Figure 116. The module `ElementsCheckMod` implements the element checking behaviour that uses the types `state` and `elements`. Here we have developed a third exclusive signature to mediate access to the module `StateMod` that specifies the function `warningChecks`, which is used by the function `checkElements`. The functions `recursionCheck` and `abstractionCheck` only have a local client, `warningChecks`, and thus are not specified in the signature. The exclusive signature `ELEMENTSSIG1` also specifies the function `elementCheck` that is used by `elementsCheck` but is hosted by a module used by `ElementsMod`.

More details regarding exclusive signature design are provided in the following section.

A.6.2 Exclusive Signature Design

Informal Description

`STATESIG` was developed to present a single signature to mediate access to `StateMod` whether as part of a partition relationship or a module use relationship. However, when implementing the system one needs more accurate information regarding the visibility requirements of clients of a module's units. This was illustrated in Section A.6.1 where three uses of the module `StateMod` were mediated by three different exclusive signatures.

In Chapter 2 we quoted Pooley and Stevens [109] definitions for abstraction and encapsulation:

> Abstraction is when a client of a module doesn't need to know more than is in the interface. Encapsulation is when a client of a module isn't able to know more than is in the interface.

We believe that an exclusive signature's rôle during analysis is to deliver encapsulation: *this is all that a client is allowed to know.* Then during design its rôle becomes the delivery of abstraction: *this is what a client needs to know.* Thus the exclusive signatures delivered in the design phase should specify a subset (upto changes enforced due to a redesign of the module architecture) of the units specified during analysis. One is specialising the interface to a module for a particular purpose.

Thus one needs to analyse the requirements of a particular use relationship or partition relationship and specify only those units in the mediating exclusive signature.

FAD Description

We refer again to Figures 114 and 116. The exclusive signatures specify that which is required for a particular relationship and no more. For example, `STATESIG3` specifies those units required by clients hosted by `ElementsCheckMod`. The exclusive signature `ELEMENTSSIG1` specifies `elementCheck` since it is used by the function `checkElements` without requiring a use relationship from `ElementsCheckMod` to `ElementMod`.

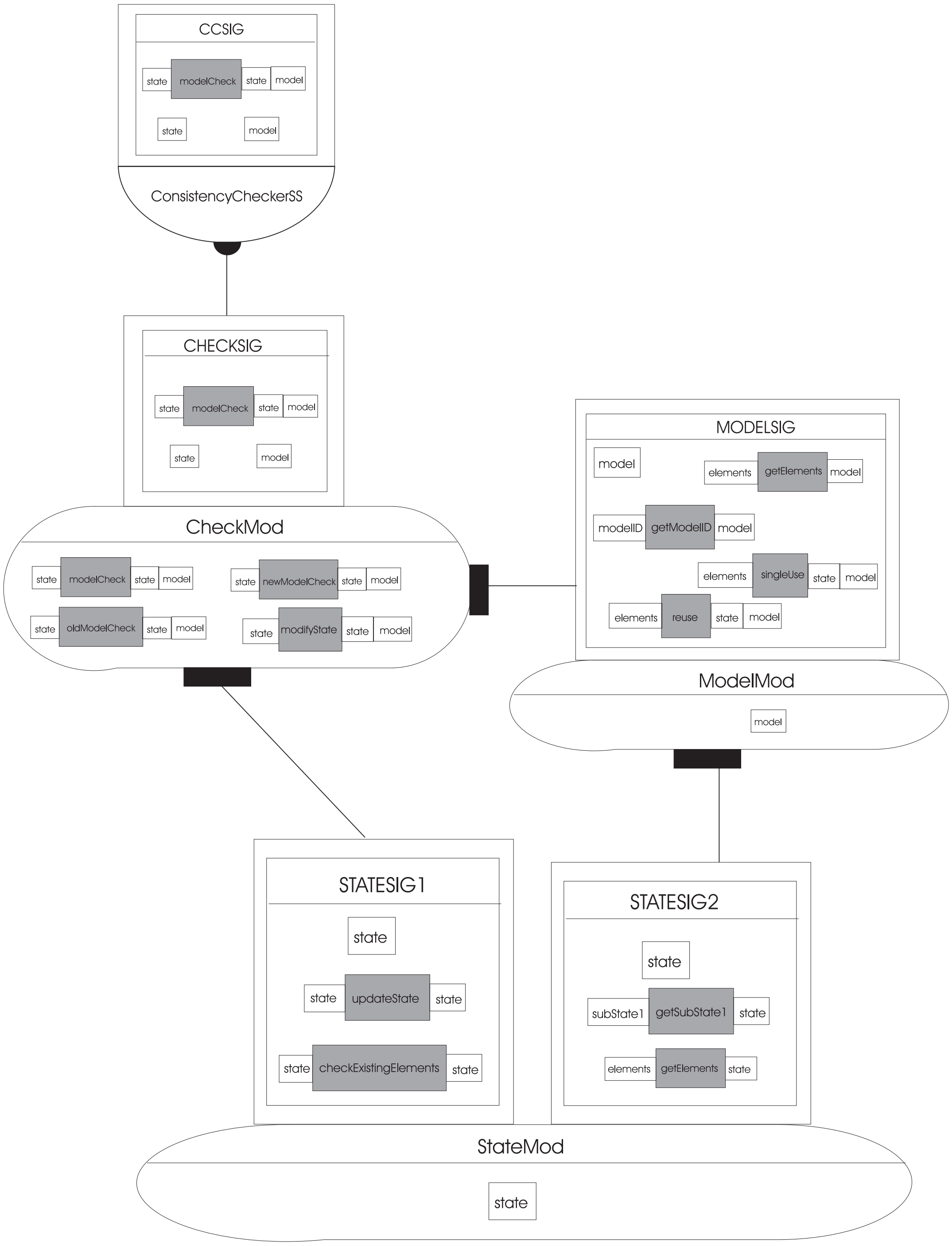

Figure 114: Module Architecture Design

Module Description Document CASE

Name:	`CheckMod`
Version:	20000722:0
Type(s):	
Permissive sig(s):	
Function(s):	`modelCheck: state -> model -> state`
	`newModelCheck: state -> model -> state`
	`oldModelCheck: state -> model -> state`
	`modifyState: state -> model -> state`
Modules used:	`StateMod : STATESIG1`
	`ModelMod : MODELSIG`
Subsystem:	`ConsistencyCheckerSS`
File:	

Description:

The module `CheckMod` hosts the functions that implement model checking functionality but does not host any types. This module therefore uses the modules which host the types `state` and `model`, but provides a single entry route into the module architecture for external clients.

Figure 115: Module Description Document for the Module `CheckMod`

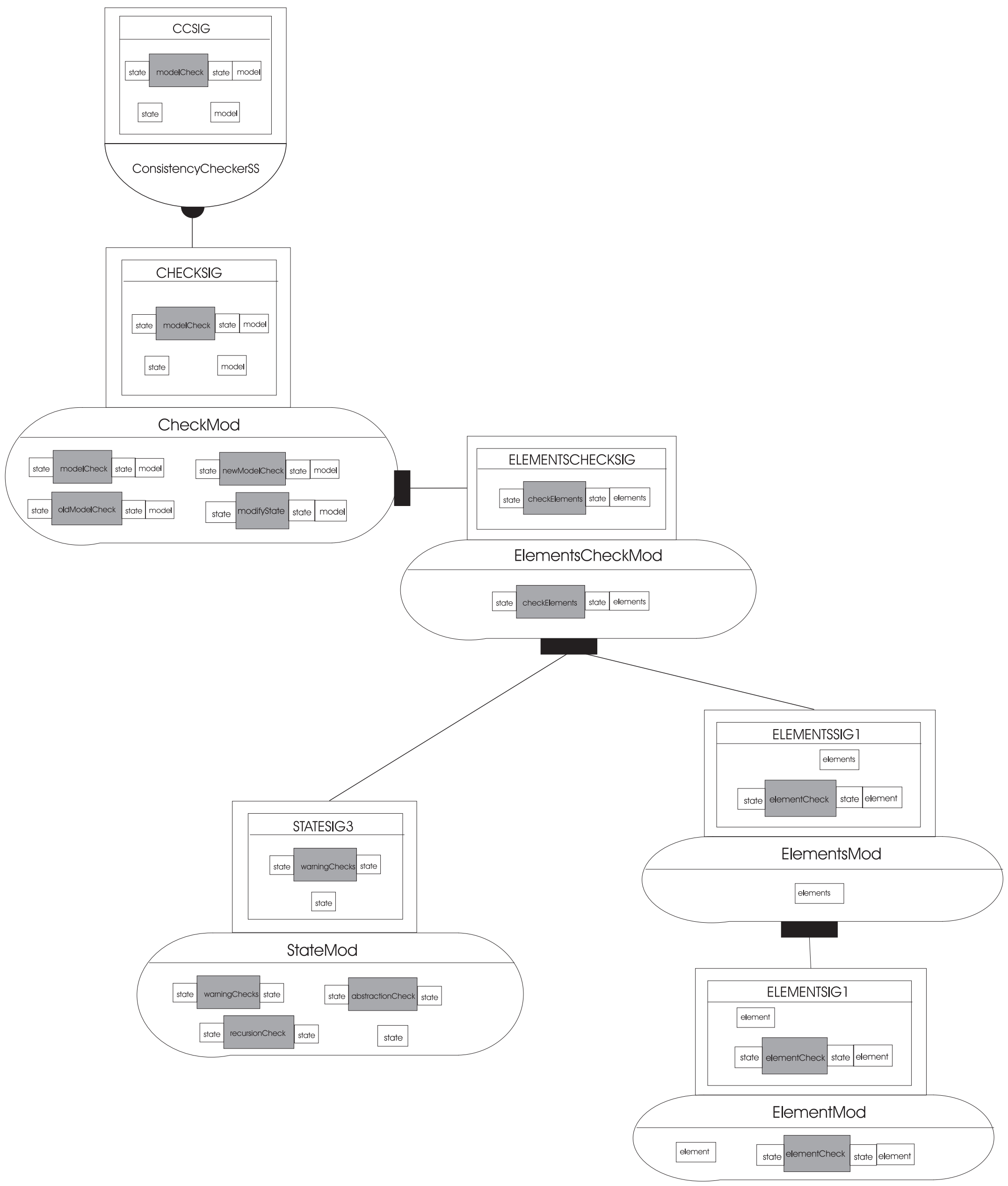

Figure 116: Another Module Architecture Design

In the following section we describe the development of the permissive signature `CONTAINERPLUS`.

A.6.3 Design of the Permissive Signature `CONTAINERPLUS`

Informal Description

The permissive signature `CONTAINER` specifies the behavioural requirements of a standard collection type. However, it does not support any behavioural requirements of the items being collected. The permissive signature `CONTAINERPLUS` inherits the specifications of `CONTAINER` but adds the flexibility required over the contained items.

That is, when an element is 'added to' or 'removed from' a collection of elements one doesn't simply update the collection with one more or one less element. When 'adding' an element one needs to test whether the element already exists in the collection. If it does then one records that the element is used in a new model. That is, one updates its model identifiers entry. If it doesn't exist in the collection then it is added to the collection.

The behaviour when 'removing' an element depends on whether the element no longer appears in any models. If this is the case then it is removed from the collection. Otherwise it remains and its model identifiers entry is updated to record its removal from a model.

FAD Description

We refer now to Figure 117. `CONTAINERPLUS` inherits from `CONTAINER` and specifies the functions `addPlus` and `removePlus`. `CONTAINERPLUS` has two parameters of kind `* ->*` and `*` respectively. `addPlus` and `removePlus` have the same type as `remove` (`addPlus` requires the item type to be an equality type) but now support behaviour specific to the instantiating element type as well as the instantiating collection type.

The type `elements` has to be updated as described in the following section, and functions over the type that used the functions `add` and `remove` will now use `addPlus` and `removePlus`.

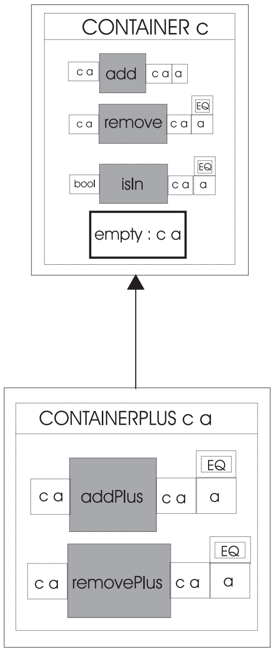

Figure 117: Design of CONTAINERPLUS

A.6.4 Design of the Type elements

Informal Description

The current design of the type elements states that it uses the types element and a collection type that must instantiate the permissive signatures CONTAINER, FOLD and FILTER. The new design includes the instantiation of the permissive signature CONTAINERPLUS by the collection type and the type element. We have decided to implement the collection type as a list, [a], since it delivers all of the required behaviour and there are no stated requirements regarding the efficiency of finding, adding and retrieving elements that would require a type such as a balanced tree.

However, the type element, the types microUnit, relationship and macroUnit, and all the types of the various sorts of micro units, macro units and relationships should be *ordered* types. This is because it will ease the discovery of existing elements both for retrieval and reuse purposes. Thus each type will instantiate the permissive signature ORD. This signature inherits the specifications of the permissive signature EQ and hence the types remain equality types as previously declared.

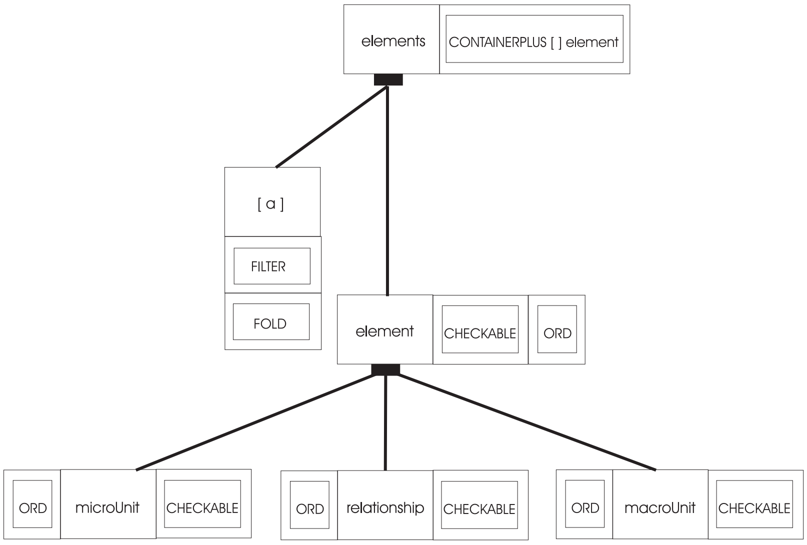

Figure 118: Design of the type `elements`

FAD Description

We refer now to Figure 118 that presents an update of the model of the type `elements`, which includes the various permissive signature instantiations described above.

In the final subsection we describe function development.

A.6.5 Function Design

Informal Description

The following `visibleFrom` and `inState` functions are used in the development of the CASE system.

```
visibleFrom : state -> function  -> function  -> bool
visibleFrom : state -> module    -> module    -> bool
visibleFrom : state -> permSig   -> type      -> bool

inState     : state -> functionUseRel -> bool
inState     : state -> moduleUseRel   -> bool
inState     : state -> typePermSigRel -> bool
```

Further scenario analyses have required `visibleFrom` functions where the second and third argument types are: `type` and `type`; `type` and `function` and so on. That is, there are several `visibleFrom` functions defined over two types used by the type `microUnit`. Each of these functions will be implemented identically since they all implement the visibility test over two micro units as described in Section A.2. They can therefore be replaced by the function

```
visibleFrom : state -> microUnit -> microUnit -> bool
```

The visibility relationship between modules is different than that between micro units and thus requires a different function. This function now requires a different name. We call it `visibleFromModule`.

The various `inState` functions can similarly be replaced by a single function whose second argument is of type `element`.

The checks of the micro unit use relationships - such as `functionUseCheck` described in Section A.4.1 - have the following operational behaviour:

1. check if the element is present in the state using `inState`; and, if not
2. check that the server unit is visible from the client unit using `visibleFrom`; and, if it is
3. add the element to the state using the function `addToState`.

We therefore replace them by a single function `microUnitUseCheck` whose second argument can be a value of type `functionUseRel`, `typeUseRel`, or `functionTypeUseRel`.

FAD Description

Various models will need to be updated to include the above changes. In the last section of the appendix we summarize the development of the subsystem and describe work to be done.

A.7 Summary

In this appendix we have presented the application of FAD to the development of a consistency checker for a CASE tool. The notation, techniques and methodological approach have been thoroughly tested.

Requirements analysis produced a collection of consistency checks many of which provide a service to the main checking of a model. Through scenario analyses and type dependency analyses we established the main set of types, their requirements and interactions, and the operational requirements of the functions that implement the consistency checks.

A module architecture was then introduced to support the development of a system based on encapsulation and abstraction. Initially we adopted a type-centric approach to module assignment that was later reviewed in light of mutual dependencies and the need for a more effective and efficient design. Exclusive signatures that mediate access to the modules were developed in tandem, and it is these that enforce the required abstraction barriers to external clients.

Developing the system to an implementation would involve:

- tailoring the design to a particular implementation language;
- implementing sections of the design and updating them based on the results of the implementation. Since the development models and their associated documentation provide a record of development, they should be updated in light of implementation experience;
- modifications due to the requirements of other subsystems. The consistency checker has been developed in isolation of the other parts of the CASE tool. Although the methodology supports an incremental approach to development, it is most unlikely that the various subsystems will simply glue together as a system free of imperfections. However, one would hope that any modifications are of a relatively minor nature and have a localised rather than widespread effect.

Bibliography

[1] H. Abelson and G.J. Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, 1985.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers – Principles, Techniques and Tools.* Addison-Wesley, 1986.

[3] S. Alpert. Primitive types considered harmful. *Java Report*, 3(11), November 1998.

[4] N. An and Y. Park. A Structured Approach to Retrieving Functions by Types. In *Proc. of the 1998 International Conference on Functional Programming.* ACM Press, 1998. Poster presented at poster session.

[5] A. Appel and T. Jim. Shrinking lambda expressions in linear time. *Journal of Functional Programming*, September 1997.

[6] J. Armstrong. The development of Erlang. In *International Conference on Functional Programming.* ACM SIGPLAN Notices, August 1997.

[7] L. Augustsson. *Haskell B. interpreter*, 1997. At ftp://ftp.cs.chalmers.se/pub/haskell/chalmers/.

[8] K. Beck and W. Cunningham. A Laboratory for Teaching Object-Oriented thinking. *Proceedings of OOPSLA – Conference on Object-Oriented Programming Systems, Languages and Applications*, 24(10):1–6, 1989.

[9] F. Belina, D. Hogrefe, and A Sarma. *SDL with Applications from Protocol Specification.* Prentice-Hall, 1991.

[10] Karen L. Bernstein and Eugene W. Stark. Debugging type errors (full version). Technical report, State University of New York at Stony Brook, Computer Science Department, 1995.

[11] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall, 1988.

[12] N.D. Birrell and M.A. Ould. *A Practical Handbook for Software Development.* Cambridge University Press, 1985.

[13] B.W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1986.

[14] G. Booch. *Software Engineering with Ada.* Benjamin/Cummings, 1987.

[15] G. Booch. *Object-Oriented Analysis and Design with applications.* Benjamin/Cummings, 1994.

[16] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[17] Object pascal language guide., 1995.

[18] J. Bosch. Design patterns as language constructs. *The Journal of Object-Oriented Programming.*, 11(2), May 1998.

[19] R. Braek. *Engineering real time systems : an object-oriented methodology using SDL.* Prentice-Hall, 1993.

[20] F. Brooks. Conceptual essence of software engineering or there is no silver bullet. *IEEE Computer*, October 1987.

[21] T. Budd. *An Introduction to Object-Oriented Programming.* Addison-Wesley, 2nd edition, 1997.

[22] R. Burgess. Designing codasyl database progams using JSP. *Information and Software Technology*, 29(3), April 1987.

[23] J. Cameron. *JSP & JSD: The Jackson Approach to Software Development.* IEEE Computer Society Press, 1989.

[24] L. Cardelli. Two-dimensional syntax for functional languages. In *Proc. Integrated Interactive Computing Systems*, pages 107–119, 1983.

[25] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4), 1985.

[26] P. Chen. The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.*, 1:9–36, 1976.

[27] P. Coad. Object-oriented patterns. *Communications of the ACM*, 35(9), September 1992.

[28] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Prentice Hall, 2nd edition, 1991.

[29] P. Coad and E. Yourdon. *Object-Oriented Design.* Prentice Hall, 1991.

[30] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method.* Prentice-Hall, 1994.

[31] The new collins compact english dictionary., 1984.

[32] Object-oriented analysis and design methods: a comparative review., 1995. At: http://wwwis.cs.utwente.nl:8080/dmrg/OODOC/oodoc/oo.html.

[33] Rational Software Corporation. Getting Started with Rational Rose., 1995.

[34] Rational Software Corporation. Rational Rose/C++., 1995.

[35] O. Dahl, E. Dijkstra, and Hoare C. *Structured Programming.* Academic Press, 1972.

[36] H. Deitel and P. Deitel. *Java How to Program.* Prentice Hall, 2nd edition, 1998.

[37] T. DeMarco. *Structured Analysis and System Specification.* Prentice-Hall, 1979.

[38] E. Dijkstra. Goto statement considered harmful. *Communications of the ACM*, 11(3):147–8, March 1986.

[39] A. Diller. *Z: An Introduction to Formal Methods.* Wiley, 2nd edition, 1994.

[40] Dominic Duggan and Frederick Bent. Explaining type inference. *Science of Computer Programming*, 27:37–83, 1996.

[41] M Eva. *SSADM Version 4: A User's Guide.* McGraw-Hill, 2nd edition, 1994.

[42] S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones. H/Direct: A Binary Foreign Language Interface for Haskell. In *Proc. 3rd International Conference on Functional Programming.*, pages 153–162. ACM Press, 1998.

[43] K. Fisher and J. Mitchell. Notes on typed object-oriented programming. In *Proc. Theoretical Aspects of Computer Software*, pages 844–885. Springer LNCS 789, 1994.

[44] D. Flanagan. *JavaScript: The Definitive Guide.* O'Reilly, 2nd edition, 1997.

[45] A flow graph applet., 1997. At: www.cogs.susx.ac.uk/users/alanje/premon/ndemo.html.

[46] K. Fowler, M. with Scott. *UML Distilled: Applying the Standard Object Modeling Language.* Addison-Wesley, 1997.

[47] M. Fowler. *Analysis Patterns: Reusable Object Models.* Addison-Wesley, 1997.

[48] FranTk, 1999. At: http://www.haskell.org/FranTk/.

[49] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[50] A. Gill and S. Marlow. *Happy: the parser generator for Haskell.*, 1995.

[51] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[52] H. Goodman. Animating Z specifications in Haskell using a monad. Technical report, University of Birmingham, April 1995.

[53] A.J. Gordon. *Functional Programming and Input/Output.* British Computer Society Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[54] D. Harel. Statcharts: A visual formalism for complex systems. *Science of Computer Programming*, 8, 1987.

[55] The HBC Compiler., 1999. At: http://www.cs.chalmers.se/˜augustss/hbc/hbc.html.

[56] T. Hopkins and B. Horan. *Smalltalk an Introduction to Application Development using Visualworks.* Prentice Hall, 1995.

[57] J. Hughes. Why Functional Programming Matters. In D. A. Turner, editor, *Research Topics in Functional Programming.* Addison-Wesley, 1990.

[58] W. Humphrey. *Managing the Software Process.* Addison-Wesley, 1990.

[59] G. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, 1992.

[60] G. Hutton and E. Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, University of Nottingham, 1996.

[61] M. Jackson. *Principles of Program Design.* Academic Press, 1975.

[62] M. Jackson. *System Development.* Prentice-Hall, 1983.

[63] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, 1999.

[64] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.

[65] M. Jones. The implementation of the Gofer functional programming system. Research Report YALEU/DCS/RR-1030, Yale University, 1994.

[66] M. Jones. Functional Programming with Overloading and Higher- Order Polymorphism. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming.*, number 925 in LNCS. Springer-Verlag, May 1995.

[67] M. Jones and J. Peterson. Hugs 1.4: The Nottingham and Yale Haskell User's System - User Manual. Research Report NOTTCS-TR-97-1, The University of Nottingham, 1997.

[68] M. Jones and J. Peterson. *Hugs 98: A functional programming system based on Haskell 98*, 1999. At http://www.haskell.org/hugs/.

[69] M P. Jones. Using Parameterised Signatures to Express Modular Structure. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1996.

[70] M. P. Jones and J. C. Peterson. Hugs 1.4. The Nottingham and Yale Haskell User's System User Manual. Technical report, The University of Nottingham, April 1997.

[71] A. Kay. The Early History of Smalltalk. In *The Second ACM SIGPLAN History of Programming Languages Conference*, volume 28(3), pages 69–75. ACM SIGPLAN Notices, March 1993.

[72] B. Kernighan and D. Ritchie. *The C Programming Language.* Prentice Hall, 2nd edition, 1988.

[73] M.J. King and J.P. Pardoe. *Program Design Using JSP – a Practical Introduction.* MacMillan, 1985.

[74] K. Laüfer. Type classes and existential types. *Journal of Functional Programming*, May 1996.

[75] X. Leroy. *The Objective Caml system.*, 1996. At http://pauillac.inria.fr/ocaml/.

[76] S. Liang, P. Hudak, and M. Jones. Monad transforners and modular interpreters. In *22nd ACM Symposium on Principles of Programming Languages*, ACM, San Francisco, Jan 1995.

[77] K. Lieberherr and I. Holland. Formulations and benefits of the Law of Demeter. *ACM SIGPLAN Notices*, 24(3):67–78, March 1989.

[78] K. Lieberherr, I. Holland, and A. Riel. Object-oriented programming: An objective sense of style. In *Proceedings of OOPSLA '88 Conference*, pages 323–334, September 1988.

[79] B. Liskov. Data abstraction and hierarchy. *SIGPLAN Notices*, May 1988.

[80] R. Malan, R. Letsinger, and D. Coleman. *Object-Oriented Development at Work: Fusion in the Real World.* Prentice-Hall, 1996.

[81] J. Martin. *Principles of Object-Oriented Analysis and Design.* Prentice-Hall, 1993.

[82] R. Martin. *Designing Object-Oriented C++ Applications using the Booch Method.* Prentice-Hall, 1995.

[83] J. A. McDermid. *Software Engineer's Reference Book.* Butterworth-Heinemann Ltd, 1991.

[84] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 2nd edition, 1997.

[85] S. Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs.* Addison-Wesley, 1992.

[86] S. Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs.* Addison-Wesley, 1996.

[87] J. Micallef. Encapsulation, reusability and extensibility in object-oriented programming languages. *Journal of Object-Oriented Programming Languages*, 1(1):12–35, 1988.

[88] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML - Revised.* MIT Press, 1997.

[89] D. E. Monarchi and G. I. Puhr. A research typology for object-oriented analysis and design. *Communications of the ACM*, 35(9), September 1992.

[90] J. Nicklisch and S. Peyton Jones. An Exploration of Modular Programs. In *The Glasgow Workshop on Functional Programming*, 1996.

[91] O'Haskell, 1999. At: http://www.cs.chalmers.se/~nordland/ohaskell/.

[92] Object faq., 1999. At http://www.cyberdyne-object-sys.com/oofaq2/.

[93] W.F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[94] Y. Park and D. Ramjisingh. Structuring Software Components based on reusability. *International Journal of Advanced Software Technology*, pages 271–290, 1995.

[95] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM.*, 15(12), 1972.

[96] L.C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 2nd edition, 1996.

[97] S. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1988.

[98] S. Peyton Jones. Multi-parameter type classes in ghc., 1998. At: http://research.microsoft.com/
Users/simonpj/Haskell/multi-param.html.

[99] S. Peyton Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell, August 2000. At: http://research.microsoft.com/users/simonpj.

[100] S. Peyton Jones et al. *Haskell 98: A Non-strict, Purely Functional Language.*, February 1999. At http://www.haskell.org/onlinereport/.

[101] S. Peyton Jones et al. *Standard Libraries for the Haskell 98 Programming Language*, February 1999. At http://www.haskell.org/onlinelibrary/.

[102] S. Peyton Jones, M. Jones, and E. Meijer. Type Classes: an exploration of the design space. In *Haskell Workshop.*, Amsterdam, The Netherlands., June 1997.

[103] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Symposium on Principles of Programming Languages*, Charleston, USA, January 1993. ACM.

[104] S.L. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell compiler: a technical overview. In *Proc. of the UK Joint Framework for Information Technology (JFIT) Technical Conference.*, 1993.

[105] Pizza, 1999. At: http://cm.bell-labs.com/cm/cs/who/wadler/pizza/.

[106] R. Plasmeijer and M. van Eekelen. *The Concurrent CLEAN Language Report (version 1.3)*, 1997.

[107] E. Poll. Subtyping and inheritance for algebraic datatypes. Unpublished, 1997.

[108] E. Poll. Behavioural subtyping for a type-theoretic model of objects. In *FOOL5: Fifth International Workshop on Foundations of Object-Oriented Languages*, January 1998.

[109] R. Pooley and P. Stevens. *Using UML: Software Engineering with Objects and Components.* Addison-Wesley, 1999.

[110] *Prograph CPX.* 200 Barrington Street, Suite 401, Halifax, Nova Scotia, Canada.

[111] C. Reade. *Elements of Functional Programming.* Addison Wesley, 1989.

[112] H.J. Reekie. *Realtime Signal Processing: Dataflow, Visual, and Functional Programming.* PhD thesis, University of Technology at Sydney, September 1995.

[113] D. Riehle and H. Züllighoven. Understanding and using patterns in software development. *Theory and Practice of Object Systems.*, 2(1):3–13, 1996.

[114] A. Riel. *Object-Oriented Design Heuristics.* Addison-Wesley, 1996.

[115] R. Rist and R. Terwilliger. *Object-Oriented Programming in Eiffel.* Prentice Hall, 1995.

[116] M. Rittri. Retrieving library identifiers via equational matching of types. In M. E. Stickel, editor, *10th Int. Conf. on Automated Deduction*, volume 449 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1990.

[117] M. Rittri. Using Types as Search Keys in Function Libraries. *Journal of Functional Programming*, 1(1):71–90, 1991.

[118] W.W. Royce. Managing the development of large software systems: concepts and techniques. In *Proc. IEEE WESTCON*, pages 1–9, 1970.

[119] J. Rumbaugh. Relational database design using an object-oriented methodology. *Communications of the ACM*, 31(4):415, April 1988.

[120] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[121] C. Runciman and N. Rojemo. New dimensions in heap profiling. *Journal of Functional Programming*, July 1996.

[122] C. Runciman and I. Toyn. Retrieving re-usable software components by polymorphic type. In *Proc. 4th International Conference on Functional Programming Languages and Computer Architecture.*, pages 166–173. ACM Press, 1989.

[123] C. Runciman and D. Wakeling. *Applications of Functional Programming.* UCL Press Ltd, 1995.

[124] D. Sanella. Formal program development in Extended ML for the working programmer. In *Proc. 3rd BCS/FACS Workshop on Refinement.*, pages 99–130. Springer Workshops in Computing, 1991.

[125] J. Sargeant, C. Kirkham, and S Hooton. *UFO1.0 Reference Manual.*, 1996. At http://www.cs.man.ac.uk/arch/people/j-sargeant/man/man.html.

[126] S. Schlaer and S. Mellor. *Object-Oriented Systems Analysis : Modeling the World in Data.* Yourdon Press, 1989.

[127] G. Schneider and J. Winters. *Applying Use Cases: A Practical Guide.* Addison-Wesley, 1998.

[128] S. Smesters, E. Nöcker, J. van Groningen, and R. Plasmeijer. Generating Efficient Code for Lazy Functional Languages. In *Proc. 5th ACM Conference on Functional Programming Languages and Computer Architecture.*, pages 592–617. ACM Press, 1991.

[129] I. Sommerville and P. Sawyer. *Requirements Engineering.* Wiley, 1997.

[130] N. Stern and R. Stern. *Structured COBOL Programming.* Wiley, 6th edition, 1991.

[131] C. Strachey. Fundamental concepts in programming languages. In *International Summer School in Computer Programming*, August 1986.

[132] B. Stroustrup. What is 'object-oriented programming?'. *IEEE Software*, 5(3):10–20, May 1988.

[133] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1994.

[134] A. Sutcliffe. *Jackson System Development.* Prentice-Hall, 1988.

[135] *TclHaskell - user manual.*, 1999. At: http://www.dcs.gla.ac.uk/ meurig/ TclHaskell/ ˜usermanual.html.

[136] A. Tolmach and A. Appel. A Debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, 1995.

[137] J. Marshall Unger. *The Fifth Generation Fallacy.* Oxford University Press, 1987.

[138] P. Wadler. Comprehending monads. In *Conference on Lisp and Functional Programming*, ACM, Nice, France, June 1990.

[139] P. Wadler. How to declare an imperative. In John Lloyd, editor, *International Logic Programming Symposium.* MIT Press, December 1995.

[140] P. Wadler. Monads for Functional Programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, number 925 in LNCS, Båstard, Sweden, May 1995. Springer.

[141] P. Wadler. Functional Programming: An angry half-dozen. *ACM SIGPLAN Notices.*, February 1998.

[142] P. Wadler. Why no one uses functional languages. *ACM SIGPLAN Notices.*, 1999.

[143] K. Waldén and J-M. Nerson. *Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems.* Prentice Hall, 1994.

[144] M. Wallace and C. Runciman. Haskell and XML: combinators for generic document processing. Technical report, University of York, 1999.

[145] J. Warmer and A. Kleppe. *The Object Constraint Language : Precise Modeling With UML.* Addison-Wesley, 1999.

[146] R. Wiener and G. Ford. *Modula-2 : A Software Development Approach.* Wiley, 1985.

[147] R. Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Computing Surveys*, 30(4), November 1998.

[148] R. Winder and G. Roberts. *Developing Java Software.* Wiley, 1998.

[149] R.J. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software.* Prentice-Hall, 1990.

[150] N. Wirth. *Programming in Modula-2.* Springer-Verlag, 1983.

[151] W. Woodman and B. Heal. *Introduction to VDM.* McGraw-Hill, 1993.

[152] E. Yourdon. *Modern Structured Analysis.* Yourdon Press, 1989.

[153] E. Yourdon and L. Constantine. *Structured Design.* Yourdon Press, 1979.

[154] A. Zaremski and J. Wing. Signature Matching: a Tool for Using Software Libraries. *ACM Transcations on Software Engineering.*, April 1995.

[155] A. Zaremski and J. Wing. Specification Matching of Software Components. Research Report CS-95-127, Carnegie Mellon University, 1997.