DIVISION OF COMPUTER SCIENCE

Inheritance, subtyping and the is-a relationship (PhD Transfer Report)

Mary Buchanan

Technical Report No.237

January 1996

Inheritance, subtyping and the is-a relationship (PhD Transfer Report)

Mary Buchanan
Division of Computer Science, University of Hertfordshire
College Lane, Hatfield, Herts. AL10 9AB

January 25, 1996

Abstract

Inheritance, subtyping and is-a are different relationships all of which have many interpretations. Moreover, definitions of each of the relationships often involve one of the other two. An attempt has been made to distinguish some of these interpretations in order to clarify the meaning of inheritance. Identification of the is-a relationship has proved particularly difficult.

Although many class-based object-oriented programming languages adopt the subtype relationship as the basis for inheritance, the F-bound relationship is found to be a better model of the evolution of type under inheritance. Future work will be directed towards establishing what effect an F-bound view of inheritance has on the is-a relationship and on the inheritance relationship in object-oriented system development as a whole.

Contents

1 Introduction

According to Wegner [1] inheritance is the property of object-oriented languages which distinguishes them from object-based languages. However, according to America [2] "inheritance is *not* essential for object-oriented languages, the object-orientedness *is* essential for inheritance". Inheritance is not a single, well-defined concept. Inheritance is often allied to subtyping but Cook [3] is a notable exponent of the opinion that inheritance is not subtyping. Inheritance is frequently used for code reuse in the absence of any subtype relationship and can be used for version control and to simulate genericity [4]. One of the uses of inheritance is to express conceptual is-a relationships such as panda is-a bear [5, page 395]. Is-a relationships may or may not coincide with subtype relationships and it is the purpose of this report to examine the is-a relationship in more detail.

An investigation into the is-a relationship inevitably means that the subtype relationship will also need to be understood. While being less heavily overloaded than the inheritance relationship, subtyping conveys more than one meaning. In programming languages the subtype relationship tends to be a syntactic relationship between method signatures. In other contexts the subtype relationship is given a semantic interpretation [6]. The data modelling community tend to use the term is-a for the concept of structural similarity [7]. In some development methods [8, 9] the subtype relationship is used to convey the notion of structural similarity between object classes. Other workers use generalization-specialization[1] to convey both structural and behavioural similarity [10, 11].

The inheritance, subtype and is-a relationships are distinguished by Lalonde and Pugh [12] in their paper entitled "Subclassing $\neq$ subtyping $\neq$ Is-a". They regard subclassing as a code sharing relationship, subtyping as a substitutability relationship and is-a as a specialization relationship. However, by the authors' own admission [12, page 62] their definition of the is-a relationship is rather imprecise.

From this brief introduction alone it is obvious that inheritance, subtyping and is-a mean different things to different people. The work outlined below in section 2 gives a more detailed account of these differences and some conclusions are drawn in section 3.

The is-a relationship is important but it is hard to identify. The subtype relationship is one possible semantic interpretation of is-a, the F-bounded relationship[2] is another. If inheritance at the language level is based on the F-bound relationship, then the semantic interpretation of specialization and generalization relationships during object-oriented system development will need to be defined.

It is the purpose of our research to ascertain what effects an F-bounded view of inheritance has on system development. An outline of the proposed future direction of the research is given in section 4.

2 Research undertaken so far

2.1 An overview of inheritance in particular and object-oriented concepts in general

2.1.1 Objects

Objects encapsulate data and methods in a single entity. They tend to be dynamic in that their data changes during their lifetime and they can be created dynamically. The data in an object is hidden from clients; it can only be accessed via the methods of the object. Those methods of an object which are externally available form the interface of the object and thereby define the object's behaviour.

[1] In this report American spelling is used for generalisation and specialisation in accordance with many of the referred texts.

[2] The F-bound was introduced [13] as a means of modelling recursively defined types such that they reflect the operational semantics of inheritance polymorphism in object-oriented langauges.

2.1.2 Classes

A class defines the structure (internal data) and the methods which all objects of the class have. Cattell [14, page 110] refers to the *intent* and the *extent* of a class. The intent defines the structure and the behaviour of objects of the class whereas the extent defines the set of objects of a particular class. The extent can be viewed as a classification of objects. Most object-oriented programming languages define the intent of a class by means of the implementation of the class. The extent of a class is not usually automatically maintained by such languages; rather it is left to a programmer to write the necessary code. It will be shown in section 2.5 that some development methods consider the extent of a class during analysis.

2.1.3 Generalization

Cattell [14, page 113] refers to generalization as the idea of a hierarchy of types of objects. Furthermore "generalization is also called inheritance, subtyping or subclassing". Cattell gives several interpretations of generalization of subtypes[3] :- specification, classification, specialization and implementation.

- Specification: Subtypes are defined as a type predicate applied to objects.
- Classification: Subtypes are used as sets to classify objects - that is to define different type extents.
- Specialization: Subtypes may add additional attributes and methods to a supertype.
- Implementation: Subtypes may provide different implementations of the methods defined in a supertype.

Cattell notes that these four views of generalization are not mutually exclusive and that in programming languages subtypes are used in combination with inheritance to provide a "convenient abstraction mechanism for all four kinds of generalization" [14, page 114]. Programming languages vary in their support for generalization and it is debatable how "convenient" the mechanisms are.

2.1.4 Inheritance

"To inherit is to receive properties or characteristics of another, normally as a result of some special relationship between the giver and the receiver" [15, page 49]. It may be the case, and often is in human society, that not all the properties and characteristics are inherited and that the properties of the receiver are not limited to those that are inherited. Many object-oriented programming languages enable such a pattern of inheritance to be reflected between classes of objects. However, if inheritance is to be tied to a relationship such as the is-a relationship, then it will be necessary to ensure that inheritance is used in a disciplined manner which maintains the given relationship. To preserve the is-a relationship it is, for example, necessary that all properties are inherited.

Within object-oriented development inheritance is regarded as being useful in two broad areas; code reuse and conceptual abstraction. Armstrong [16] has coined the terms C-inheritance and A-inheritance to distinguish the two uses.

According to Cook and Palsberg [17, page 434] "inheritance is a mechanism for differential, or incremental, programming. Incremental programming is the construction of new program components by specifying how they differ from existing components". Cook and Palsberg also say [17, page 433] that "the most precise and widely used definition of inheritance is given by the operational semantics of object-oriented languages. The canonical operational semantics is the 'method lookup' algorithm of Smalltalk:

> When a message is sent, the methods in the receiver's class are searched for one with a matching selector. If none is found, the methods in that class's superclass are searched next. The search continues up the superclass chain until a matching method is found.

[3] Cattell defines type to be the intent of a class and as such it includes the structure (the attributes and the relationships in which objects of the type can participate) and the behaviour (the methods associated with the type).

> When a method contains a message whose receiver is `self`, the search for the method for that message begins in the instance's class, regardless of which class contains the method containing `self`.
>
> When a message is sent to `super` the search for a method begins in the superclass of the class containing the method. The use of `super` allows a method to access methods defined in a superclass even if the methods have been over-ridden in the subclass.

Unfortunately, such operational definitions do not necessarily foster intuitive understanding".

The fact that self-references in an inherited method are linked to the class which is inheriting the method (rather than remaining linked to the class in which the method is defined) leads Cook and Palsberg [17, page 434] to state that "the essence of inheritance" is that "it is a mechanism for deriving modified versions of recursive functions".

Wegner [18] responds to the question "What is the essence of inheritance?" [18, page 74] as follows "Inheritance is a mechanism for sharing and reusing behavior (sic). It is distinguished from other behavior sharing mechanisms by delayed binding of self-reference so that superclasses may merge their identity with the subclasses that inherit them".

Cook and Palsberg have developed a denotational semantics for inheritance which they admit may be considered more complex than the operational semantics because it requires an understanding of fixed-points [17, page 441]. However, they consider that the main advantage of their denotational semantics is that "it suggests that inheritance may be useful for other kinds of recursive structures, like types and functions, in addition to classes". They state that another advantage is that they have shown that although inheritance is a natural extension of existing mechanisms, it "does provide expressive power not found in conventional languages by allowing more flexible use of the fixed-point function".

Inheritance in object-oriented programming languages is the mechanism that enables code reuse to take place. There are many advantages of using C-inheritance [16, page 23] and a number of these are outlined below:

- a reduction in the amount of code necessary to implement a program
- the ability to extend and alter programs without affecting versions of the program which depend on the old code
- the ability to broadcast code extensions to all members of an inheritance hierarchy
- the ability to develop heterogeneous collections.

In addition C-inheritance can sometimes be used to implement A-inheritance. According to Armstrong [16, page 7], A-inheritance is used in conjunction with the is-a relationship. During analysis entities in the problem domain can be related by is-a relationships such that conceptual hierarchies of the form Robin is-a Bird are developed. It is obvious, that central to this process is a sound understanding of the meaning of the is-a relationship.

2.2 The is-a relationship

It has been claimed that analysis should be aimed at capturing relationships in the problem domain [19]. One such relationship is the is-a relationship which aims to reduce complexity and redundancy [4].

The is-a relationship has its origins in semantic data modelling and consequently the emphasis of the relationship naturally depended on data rather than behaviour. This is at odds with objects in which the data is encapsulated and behaviour is the predominant view. What needs to be clarified is whether the is-a relationship is defined by both data and behaviour. Other matters which need to be addressed include the nature of the relationship itself. Is the relationship to be one of set inclusion? Does the is-a relationship encompass substitutability? Is specialization the main purpose of the relationship? How well can object-oriented programming languages implement a given is-a relationship?

Regarding semantic data modelling, Hull [7] describes the Generic Semantic Model (GSM) using the World Traveler (sic) Database as an example. PERSON describes an abstract data type such that a set of objects of type PERSON are associated with the type. Each object has a unique identifier. There are three subtypes of PERSON namely TOURIST, BUSINESS-TRAVELER and LINGUIST which are related to PERSON by the is-a relationship. The interpretation of is-a is that subsets of the set of PERSON identifiers would be associated with the subtypes and these subsets may overlap. The subtypes inherit all the attributes of type PERSON and add attributes of their own. The attributes are all structural (name, address for example).

Subtypes may be *derived* or *user-specified.* A derived type can be formed by evaluating stored data; for example, the set of LINGUISTS is derived from the set of PERSONS by evaluation of the PERSON attribute storing the languages spoken (a person speaking two or more languages is a linguist). The subtypes TOURIST and BUSINESS-TRAVELER are user-specified; a person can only be in one of these sets if declared to be so by the user.

The subset is-a relationship described by PERSON and its subtypes can be regarded as representing a specialization relationship. In contrast to this, Hull notes that semantic schema also use is-a to create a new type from the union of types. For example, a VEHICLE type may be formed from the union of the types CAR, BOAT and PLANE. The is-a relationship described by such a union of types is one of generalization. The generalized type is *partitioned* by its disjoint subtypes. (Non disjoint generalization is also possible.) In contrast, specialized types are *contained* in their super type. The specialized types need not be disjoint. The update semantics of the two is-a relationships are different. In the specialization is-a, deletion from a subtype has no effect on the supertype whereas in the generalization is-a, deletion from a subtype requires deletion from the supertype.

RM/T is a relational model extended with semantic schema constructions [7]. In this model, *alternative generalization* is used to form subsets of union types. For example, the type CUSTOMER could be defined as an alternative generalization of PERSON, BUSINESS and PARTNERSHIP. A customer must be either a person or a business or a partnership but the set of customers does not have to contain *all* the persons, businesses and partnerships in the system.

The above discussion has identified three is-a relationships:

- Specialization is-a
- Generalization is-a
- Alternative generalization is-a.

Brachman [20, page 30] claims that there are almost as many meanings for the is-a link as there are knowledge-representation systems. Brachman categorises this myriad of relationships under two main headings; *generic/generic relationships* and *generic/individual relationships.*

Generic/generic relationships

1) Subset/superset.

If nodes represent sets, then the is-a relationship between nodes represents the subset relationship. For example, if a Tourist is-a Person, then every t which is a member of Tourist is also a member of Person.

2) Generalization/specialization.

Generalization is taken to mean a relation between predicates. The relation is expressed as the material conditional (if...then). For example, if Person(p) is a predicate taken to be a generalization of Tourist(p), then an is-a between them means that for every entity p if Tourist(p) then Person(p).

3) A Kind Of (AKO)

This is similar to generalization. The difference is that whereas generalization relates arbitrary predicates, AKO implies that the nodes are limited to representing "kinds". The example Brachman gives is that camel is a kind of mammal.

4) Conceptual containment

The is-a relation is used to indicate that one description includes another. For example, to be a triangle is to be a polygon with three sides.

5) Role value restriction

The example given by Brachman is "the trunk of an elephant is-a cylinder 1.3 meters long". The relationship is not taxonomic since the relationship is not between a type and a subtype.

6) Set and its characteristic type

When used in this sense, the is-a relationship conveys the relationship that exists between the concept of an elephant and the set of all elephants. The relationship is not taxonomic.

Generic/individual relationships

1) Set membership

If the generic is a set, the is-a relationship is one of set membership. For example, Fred is-a Person means that Fred is a member of the set of persons.

2) Predication

If the generic is Person and the individual is Fred, then the is-a relationship expresses the fact that Person(Fred).

3) Conceptual containment

The individual node is considered to be a structured description and the generic is used to help construct the description. For example, the individual could be "the king of France" and the generic could be "king".

4) Abstraction

In this relationship, the generic type is abstracted into an individual (the reverse of conceptual containment). For example, "the eagle is an endangered species". The individual is The Eagle and the generic is Eagle.

Default relationships

Brachman considers that the most widespread use of is-a is as a default relationship. A bird is assumed to be able to fly but the property may be cancelled to accommodate ostriches. The ability to cancel properties is seen as an advantage in that it enables exceptions to be represented. However the semantics of such default relationships are hard to predict. Brachman points out that one of the consequences of the default view is that nodes cannot be thought to represent the concepts that their names suggest. The nodes can only be thought of as place holders for bundles of default properties because of the one-way nature of the default is-a. Using Brachman's example to illustrate, if Clyde is-an Elephant then he will have the properties of Elephants. However, if Clyde has the properties of Elephants it is not possible to say that he is-an Elephant because he could be almost anything. He could be a Giraffe with all typical giraffe properties cancelled and with exactly those properties of elephants remaining/added. Brachman elaborates on this theme in his paper "I Lied about the Trees" [21]; his example of a two toed sloth is appealing.

Is-a and inheritance

Brachman notes that although is-a and inheritance are often considered together, his classification of is-a relationships is unrelated to inheritance. In his view, inheritance is an implementation technique which localises information to reduce storage requirements. Is-a relationships are a means of representing knowledge; the implementation of such relationships may or may not involve inheritance. For example, the is-a relationship could be implemented by repeating all the properties of the supertype in the subtype; this might turn out to be more efficient than searching through long inheritance hierarchies. On the other hand is-a relationships could be inferred independently of an inheritance classification defined by the user.

Armstrong [16] and [4] considers the use of inheritance in modelling is-a relationships. He bases his interpretation of is-a on Brachman's generic/generic subset definition and emphasises that

attributes (what Brachman calls properties) are inherited. Since the properties are represented as data, Armstrong considers that Brachman's is-a is mainly structural (rather than behavioural). Brachman himself points out [20, page 33] that when is-a is based on predicates there is no indication of structure but that when the objects related by is-a are intended to be concepts or descriptions, then structure is carried between descriptions. Is-a relationships can also be based on behaviours as well as structure and Armstrong illustrates [4, page 20] how different hierarchies might result from structure-oriented and behaviour-oriented approaches.

Armstrong claims [4, page 20] that strict is-a relations are attractive interpretations of inheritance for object-oriented modelling. (Strict is-a relations are those in which the subtype has all the properties of the supertype; default is-a relations are not strict.) The reasons given for this attractiveness are that:

- Is-a offers a modelling abstraction for organising perceptions about the structure of a problem domain
- Is-a relations are domain-independent and can therefore be applied to the analysis of any problem
- Is-a relations could be formalised
- Is-a relations can reduce redundancy of information by capturing commonalities of data structure and behaviour between system components and thereby...
- ...is-a relations make systems easier for humans to understand.

Armstrong also claims that is-a relations can be mapped into inheritance as provided by object-oriented programming languages. While this is certainly true for some is-a relationships, it is not universally true and we will return to a discussion of this later in section 2.4.

While discussing the use of inheritance by programmers, Armstrong states "our experience is that arguments about the proper use of inheritance which do not start from a precise definition of is-a usually turn out to be disguised arguments for haphazard inheritance". It is our contention that there is a need for analysts, designers and programmers to agree on precise definitions of is-a. It will be necessary to consider how models based on structural data is-a relationships can be refined into behavioural is-a relationships; contravariant/covariant considerations [22] can make this more difficult than might first be apparent. It may be that models should be based on behavioural rather than structural is-a relationships

Armstrong [16, page 12] notes that since is-a relates abstract descriptions of sets of entities, identifying is-a relations involves deciding what properties are of interest in the problem domain. As such, is-a relations are not intrinsic to a problem domain but are imposed by domain modellers. The entities that can be related by is-a relations must be understood before the is-a relation in question can itself be understood. Armstrong gives the following examples:

- Subtyping interprets is-a as a relation between types and the subsets of values described by the types. Armstrong's examples are confined to subranges of numeric types. The same set of operations is valid for the type and its subtype.
- Abstract classes interpret is-a as a relation between sets of individuals or examples rather than mathematical values. The examples have attributes which are used to determine set membership.

Rumbaugh [23] discusses incorrect uses of inheritance and states that most of these can be avoided if the following rule is adopted:

- B is a subclass of A if you would say B is an A

He goes on to say that the subclass should be fully compatible with its superclass in every respect and that inheritance should only be used when you want to inherit *all* the properties of a superclass. An example he gives is that an Apple Orchard is an Orchard but he does not show the structure or the behaviour of the classes. He does not define the is-a relationship in any further detail.

2.3 Types and their relationships

Whereas monomorphic type systems constrain values to belong to only one type, polymorphic type systems permit values to belong to more than one type. A classification of polymorphism due to Cardelli and Wegner [24] elaborates that due to Strachey [25] such that universal polymorphism is subdivided into parametric and inclusion polymorphism and ad hoc polymorphism is subdivided into overloading and coercion.

A discussion of types and some of the relationships between them is given by Buchanan in [22]. The most well known of these type relationships is subtyping, a form of inclusion polymorphism. Other type relationships are the F-bound relationship and the type class relationship due to Wadler and Blott.

2.3.1 The subtype relationship

The subtype relationship ensures that a value of a subtype may be used in any context which requires a value of a supertype. Cardelli and Wegner [24] have identified rules for record subtyping a summary of which is given in [22] and in [26]. An important rule required to maintain the subtype relationship is that a subtype may add monotonically to the fields or functions provided by the supertype, it may never delete any. Function subtyping requires that an argument type of a subtype function must be a supertype of the argument to the supertype function (contravariance) whereas the result type of a subtype function must be a subtype of the result type to the supertype function (covariance). The contravariance rule for function argument types guarantees that a subtype instance can be used anywhere that a supertype instance is specified.

These rules are based on the signatures of types and need to be augmented with axioms for invariants, preconditions and post conditions if the semantics of types are to be considered in the subtype relationship. Simons [26] defines such axioms.

In many object-oriented programming languages (Eiffel, C++, Modula-3, Oberon-2, Trellis/Owl) the class implementation inheritance hierarchy is merged with the subtype hierarchy. However, not all such languages obey all the subtype relation rules [26, page 9]. Eiffel breaks the contravariance rule but version 3.0 uses system wide checking to determine whether any unsafe assignments have been made. C++ and Modula-3 have a no-variance rule rather than contravariance whereas Trellis/Owl obeys the contravariance rule. Oberon-2 obeys the subtype rules by the simple if constraining rule that function signatures are not permitted to change in subclasses. Eiffel supports axioms for invariants and pre- and post conditions. Through the use of the *private* inheritance construct, C++ permits a class to inherit implementation without having to maintain the subtype relationship but since the subtype and inheritance hierarchies are combined, a class cannot claim to be a subtype of a class which is not an ancestor.

The contravariant rule can be used in object-oriented programming languages to prevent the possibility of dynamic type errors [27]. However, the contravariant rule places severe limitations on the ability of the subtype relationship to support specialization. As an example, inspired by [23], consider a type Orchard with a subtype Apple Orchard and a type FruitTree with subtypes AppleTree and PearTree. Suppose Orchard has an operation `addTree` which takes a FruitTree as argument. To specialize the AppleOrchard we would like `addTree` to take AppleTree as argument but we cannot do this because AppleTree is not a supertype of FruitTree which it needs to be if the contravariant rule is not to be broken. To preserve contravariance, we would have to make FruitTree a subtype of AppleTree which is opposite to our specialization classification of trees. Alternatively, we could leave `addTree` in AppleOrchard so that it took FruitTree as an argument. Since AppleTree is a subtype of FruitTree we could still add AppleTrees to our AppleOrchard but unfortunately we could also add PearTrees as well which is not what we want. A more detailed example concerning Employees and Salaried Employees is given by Armstrong [16, page 114].

When the implementation and the subtype hierarchies are combined, problems arise when subclasses derived by implementation inheritance are not in a subtype relation. America [2, 28] and Porter [29] overcome the limitation of restricting subclasses to subtypes by separating the subtype hierarchy from the implementation hierarchy. America has developed the language POOL

(Parallel Object-Oriented Language) [2, 28] in which inheritance is used for code reuse and types are used "to base our conceptual specialization hierarchy on" [2, page 239]. In POOL a type is defined as "a collection of objects that share the same externally observable behaviour" whereas a class is defined as "a collection of objects that have exactly the same internal structure" [28, page 161]. The criteria for deciding whether an object belongs to a type depend not only on the individual messages an object responds to but also on which order the messages are received. In order to specify the behaviour of the methods America would like to have a specification language with "at least the power of first-order predicate logic" [28, page 163]. Since it is "inherently impossible for a compiler to check the subtyping relationship if the specification language is so powerful" [28, page 163], POOL augments the specification of a type with *properties* which are just identifiers. For example, type Stack would have the property LIFO. The compiler cannot check that an implementation of Stack has correctly implemented the LIFO property, the property is used solely to determine whether one type is a subtype of another. The first rule in America's definition of subtyping [28, page 164] states that the object properties of the supertype must be among those of the subtype. The second rule covers contravariance of argument types and covariance of result types.

Although the separation of inheritance and subtyping in POOL results in more flexible code reuse, the contravariant rule will still give problems when the subtyping hierarchy is used to express "conceptual specialization". An advantage of separating inheritance and subtyping is that heterogeneous collections can be created from objects which are not related by inheritance [28, page 168]. The objects need have only part of their behaviour in common; for example, the objects could all have the method `print` as defined in the type `Printable`. The heterogeneous collection would be defined to take type `Printable` and could thus contain any object whose type was a subtype of the type `Printable`, that is any object having the method `print`.

Porter [29] has developed the language Portlandish such that the type and implementation hierarchies are separated. Types are related by subtyping and are expressed as signatures only; semantic behaviour is not specified. Implementations (used instead of classes) are related by inheritance. The subtype hierarchy is explicitly specified by the programmer; it is not inferred by the compiler. "The intent of the subtype relationship is to realize the *principle of substitutability* and, in particular, the weak form of the principle" [29, page 22]. For the weak principle of substitutability to hold, the substitution of a type by a subtype must not result in runtime errors. The strong principle of substitutability requires that when a subtype is substituted for a type, the behaviour of the program will not change. Types in Portlandish are concerned with protocol only, not with semantic behaviour, hence the emphasis on the weak form of the principle of substitutability. The implementation hierarchy is also specified by the programmer. Porter expects [29, page 25] that the two hierarchies will usually be isomorphic but gives the following examples of situations in which this will not be the case: a type may be abstract in which case it will not have an implementation, a type may have more than one implementation, an implementation may apply to more than one type, types which are not related by subtyping may have implementations related by inheritance and finally the type and implementation hierarchy may run in opposite directions.

Portlandish adopts an interesting approach to exact typing; if a variable has type Person it can refer to objects of type Person or to objects of subtypes of Person, if a variable is qualified by an implementation of a type such as PersonImpl then it can only refer to an object which is implemented by PersonImpl.

As in POOL, the typing hierarchy in Portlandish is still governed by the subtype relationship. Where desired specializations would result in problems due to the contravariance requirement, it seems likely that hierarchies will have to be based on abstract types. For example, rather than have B as a subtype of A, an abstract type G will be created such that A and B both inherit from G. (This requires foresight.)

Simons et al. [30] consider that separating the subtype hierarchy from the implementation hierarchy restricts inheritance to a subsidiary role (for code sharing only). They take the view that subtyping is "too impoverished a type model to cope with what really happens under inheritance" [30, page 10]. Instead they propose using the F-bound relation as a type model for inheritance

and have developed the language BRUNEL [31, 32] based on this concept.

2.3.2 The F-bound relationship

Canning et al. [13, 3] introduced the concept of the F-bound relationship (summarised by Buchanan in [22] and by Simons in [26]) to overcome the limitations of the Cardelli and Wegner record subtyping model in defining recursively defined types. The inability of the record model to define recursive types means that the model does not reflect the operational behaviour of object-oriented programming languages in those situations where functions are being invoked polymorphically.

The F-bound relationship is based on the concept of *type inheritance* which ensures that a subclass created by implementation inheritance, while not necessarily being in a subtype relationship with its superclass, has nevertheless inherited the type of its superclass. The advantage of type inheritance is that implementation inheritance can be used to portray specialization, without the constraints imposed by contravariance, and static type checking is not compromised.

The motivation for the F-bound is given briefly here but a more detailed explanation can be found in [22, 26]. An excellent tutorial on object-oriented type-theory including the F-bound relationship is given by Simons in [33].

Consider the behaviour of a moveable planar point type:

```
interface Point
        x: Real
        y: Real
        move(Real, Real): Point          /Positive recursion/
        equal(Point): Boolean            /Negative recursion/
```

Suppose we wish to inherit from Point to define a ColourPoint type. Under the record subtyping model the resultant interface for ColourPoint would be:

```
interface ColourPoint
        x: Real
        y: Real
        move(Real, Real): Point
        equal(Point): Boolean
        colour: Colour
```

The *move* operation can be applied to a ColourPoint, due to the subtype relation between Point and ColourPoint, but the type returned would be a Point not a ColourPoint as required to maintain the colour attribute. Likewise, the *equal* operation permits two ColourPoints to be compared for equality but also allows a Point and a ColourPoint to be compared which might not be what is desired.

What is required is that the operations inherited from Point have any references to Point changed to ColourPoint. This is achieved if inheritance of types takes place under the F-bound model. The record subtype model considers Point to be a *bounded type* which can receive the ColourPoint type, that is Point is an actual most general type. The F-bound model considers Point as a *type-bounded parameter*, that is as a syntactic abbreviation for an inclusive (non-disjoint) set of types [26, page 18]. Under this interpretation, the functions *move* and *equality* are not treated as belonging to the type Point but rather as belonging to some space of types constructed from Point. The types are constructed using a parameter which satisfies the bounds imposed by Point. In order for this to happen, the Point interface is *generalised* by defining a type function, `F[t]`, so that it can apply to any type `t`.

```
F[t] =  {x: Real,
         y: Real,
         move: Real, Real -> t,
         equal: t -> Boolean}
```

ColourPoint can be defined using type inheritance as:

```
interface ColourPoint
        inherits Point
        colour: Colour
```

The type inheritance defined by ColourPoint is regarded as an extension of the type function `F[t]` and is expressed by the type function `G[t]`:

```
G[t] = F[t] + {colour: Colour}
```

When we bind `t` to `ColourPoint` in `G[ColourPoint]`, then all references `t` in `F[t]` are also bound to `ColourPoint`:

```
interface ColourPoint
       x: Real
       y: Real
       move(Real, Real): ColourPoint
       equal(ColourPoint): Boolean
       colour: Colour
```

A subtype relation holds between the *type functions*. For all `t`, `G[t]` $\leq$ `F[t]`. For example, for type `Point`, `G[Point]` $\leq$ `F[Point]`. The functions of type `Point` can be invoked by any type such that `t` $\leq$ `F[t]`. The functions of type ColourPoint can be invoked by any type such that `t` $\leq$ `G[t]`. Since `G[t]` $\leq$ `F[t]` we have that `t` $\leq$ `G[t]` $\leq$ `F[t]`. In general, any type satisfying `t` $\leq$ `G[t]` also satisfies `t` $\leq$ `F[t]`. Hence we have that since ColourPoint satisfies `t` $\leq$ `G[t]` it also satisfies `t` $\leq$ `F[t]` and consequently functions defined for Point can also be invoked for objects of type ColourPoint [22, page 14].

To summarise, the types Point and ColourPoint both satisfy the same F-bound, since Point $\leq$ F[Point] and ColourPoint $\leq$F[ColourPoint], but Point and ColourPoint are not in a subtype relation with each other since neither Point $\leq$ ColourPoint nor ColourPoint $\leq$ Point.

A single parameter, `t`, precludes the possibility of mixed-type calls because a single parameter cannot be uniformly instantiated by a mix of types. For example, it would not be possible to compare a Point type and a ColourPoint type for equality. Such a comparison would obviously be restricted to a comparison of coordinates only, colour would have to be ignored. There might be circumstances in which colour is unimportant and it would be desirable to permit such a comparison. This can be achieved by declaring in the polymorphic typing function all of the parameters that are to be included in the F-bound:

```
F-Point [t , u] =
         {x: Real,
          y: Real,
          move: Real, Real -> t,
          equal: u -> Boolean}
```

The parameters t and u may be instantiated by distinct types satisfying the F-bound or by the same type. Hence a Point and a ColourPoint could be compared for equality of x and y coordinates.

Point and ColourPoint can be considered to be in a "classification" relationship in that ColourPoint is a specialization of Point. The F-bound concept could also be used to define interfaces for use in unrelated classes. For example, `F-Iterator[t]` could contain iterator functions which could be inherited by LinkedLists, PriorityQueues and Sets:

```
G-LinkedList[t] = F-Iterator[t] + ...

G-PriorityQueue[t] = F-Iterator[t] + ...

G-Set[t] = F-Iterator[t] + ...
```

The use of such interfaces would enable context-dependent [4, page 21] heterogeneous lists to be created without the need to create inheritance relationships between the types in the list. Multiple inheritance of F-bounds enhances the use of common interfaces.

The ability of F-bound type inheritance to model the operational semantics of object-oriented programming languages has prompted Simons to remark that "strict inheritance is therefore a much grander notion than some authors might lead you to believe - subtyping among parameterised type-constructors, interpreted formally as subsumption among categories. Inheritance is principally a relationship among type-constructors, extending finally to actual types" [26, page 22].

Several other theories of type inheritance have been proposed - for references see [3, page 126] and [3, page 133] - but as yet we have not explored these.

2.3.3 The type class relationship

Wadler and Blott [34] have proposed another way of quantifying types without involving a subtype relationship. They define *type classes* in order to provide a unified approach to the typing of both overloaded operators and functions defined in terms of the operators. Their ideas are outlined below and a more detailed account is given by Buchanan in [22].

A type class declares the names and signatures of functions which are expressed in terms of a type variable which ranges over every type. To belong to a type class, a type must have functions defined with the same names and with appropriate types. Consider the type class, `Num`, taken from Wadler and Blott [34, page 63]:

```
class Num a where
          (+), (*)  ::  a -> a -> a
          negate ::  a -> a
```

The class declaration states that a type may belong to class `Num` if it has functions (+), (*) and *negate* which are bound to the appropriate types. In order to assert that a type such as `Int` belongs to the class `Num` an instance declaration must be made:

```
class Num Int  where
          (+)  =  addInt
          (*)   =  multInt
          negate  = negInt
```

It is assumed that *addInt*, *multInt* and *negInt* have been defined on integers; if not then the definitions would occur in the type instance declaration. The type system used is based on that of Hindley/Milner [34] which means that the types of declarations can be inferred. For `NumInt`, the type inference algorithm must verify that the bindings of the operation names to the operation definitions do have the appropriate type. For example, *addInt* should have type `Int -> Int -> Int`.

Another instance, the type `Num Float`, of the class can be declared as:

```
class Num Float  where
          (+)  =  addFloat
          (*)   =  multFloat
          negate  = negFloat
```

If (+) is applied to integers, the *addInt* code will be invoked whereas if it is applied to reals, the *addFloat* code will be invoked. This is analogous to conventional overloading in languages such as Standard ML. The benefits of type classes become apparent when functions defined in terms of overloaded operators are considered. Functions can be defined for a class such that the functions can be applied to any type which is a member of the class. For example, consider the function *square* defined for the class `Num`:

```
square (x) = x  (*)  x
```

The type of *square* can be inferred as:

```
square  :: Num a  => a -> a
```

That is, *square* has type `a -> a` for every `a` that belongs to class `Num`.

It is possible to apply *square* to integers and to reals whereas in Standard ML it is not possible to define functions in terms of overloaded operators. In other approaches to overloading, a function such as *square* could be written but it would not have a parametric polymorphic type; rather it would stand for two overloaded versions of *square* with types `Int -> Int` and `Float -> Float`. This may not seem much of a problem but exponential growth of overloaded functions can soon occur. For example, consider the function *squares*:

```
squares  (x, y,  z)  =  (square x,  square y, square z)
```

The parameters x, y and z may each be typed as `Int` and `Float` which gives rise to eight possible types for *squares*. The type class approach with its inferred typing is obviously much simpler and more elegant than the explicit overloading approach.

The functionality of type classes can be extended by subclassing. Suppose a type class `Eq` is defined to express equality as in [34]:

```
class Eq  a  where
          (==)  ::  a -> a -> Bool
```

with instances:

```
instance  Eq  Int  where
          (==)  = eqInt
```

```
instance  Eq  Char where
          (==)  = eqChar
```

If `Num` is made a subclass of `Eq`, then every type which belongs to `Num` will have the equality operator defined on it:

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

This type declaration asserts that `a` may belong to class `Num` only if it also belongs to class `Eq`. This means that the instance declaration `Num Int` is only valid if `Int` has been asserted to belong to class `Eq`, this will only be true if there is an instance declaration `Eq Int` active within the same scope as `Num Int`. If a function is defined over the type `Num a`, then the qualifier `Eq` is implied and the equality operation can be used in the function definition.

A type class may have multiple superclasses and subclasses.

Subclasses enable the functions of one type class, for example `Num`, to be concatenated with the functions of another, such as `Eq`. Functions cannot be redeclared and a form of strict inheritance occurs.

However, types related by the fact that they are members of the same type class cannot be substituted for one another; for example, the function *equal* cannot be applied to a variable of type `Point` and a variable of type `ColourPoint` since *equal* is only defined for arguments of the same type.

Whereas subtype and F-bound relationships enable the *same* code as well as different code to be invoked by related types, the type class concept results in *different* code being invoked by the types within a type class.

A type class can be thought of as an abstract data type which defines a syntactic type interface. Instance declarations define implementations and thereby have semantics. It would be possible for different instance declarations to have operations with the same name but different semantics since the only requirement is that the signatures are the same. Wadler and Blott consider the possibility of adding assertions to the class type declarations in order to specify the semantics of operations [34, page 68].

2.4 Inheritance in programming languages

2.4.1 Inheritance related to subtyping

Most object-oriented programming languages (including C++, Modula-3, Oberon-2 and Eiffel) automatically regard subclasses as type-compatible with their superclasses. Type-checking rules are based on the signatures of the operations for the types and consequently do not address the semantics of the operations. An operation may type check satisfactorily but have totally changed its behaviour from that defined for the same operation defined on a superclass. The fact that type substitutability is guaranteed does not therefore guarantee that behavioural substitutability is guaranteed. Thus a client which expects a certain behaviour from a superclass object may find that if a subclass object is substituted, the behaviour changes. At present disciplined design practices are relied on to prevent incompatible behavioural redefinition [35].

Meyer [36] advocates the use of programming by contract in order to guarantee behavioural substitutability when using inheritance. Pre-conditions are used to indicate the conditions under which an operation can be used, post-conditions describe the conditions that are guaranteed after an operation has been invoked. Contracts also specify the invariants maintained by an implementation. Subclasses fulfil the contracts of their superclasses and accordingly strict behavioural subtype relationships can be maintained, see Armstrong [4, page 23]. Mitchell et al. [37] use the term *behavioural type* to describe types which include some idea of behaviour expressed by

contracts (as opposed to types described only by a set of signatures without any concept of behaviour). They also make the point that if assertions (pre- and post-conditions and invariants) are to be executable, then they cannot contain logical quantifiers. As a result, an assertion such as "for all x in set S, property P(x) holds" has to be simulated by a function that iterates over a data structure. Among other problems associated with contracts, they note that post-conditions and invariants sometimes have to be expressed in terms of class attributes which are not part of the public view of a class [37, page 5].

Some salient features of inheritance in the languages Eiffel and C++, which support multiple inheritance, and Oberon-2 and Modula-3, which support single inheritance, are given below. A comparison of these languages can be found in Mayes [38] and of all but Modula-3 in Mayes and Buchanan [39]. All of the languages are statically typed. The languages use different terminology to refer to the method and data fields of an object; in what follows we use features (as in Eiffel) to refer to both methods and data attributes.

C++

C++ is defined in [40] and described in [41]. A detailed account of inheritance in C++ is given by Buchanan in [42]. The points relevant to this report are noted here.

Although C++ combines inheritance with subtyping, the use of *private* inheritance does permit the reuse of code without subtyping. C++ functions are statically bound at compile time unless they are declared as *virtual* functions. Virtual functions are dynamically bound and enable inheritance polymorphism to be effected. Virtual functions can be redefined[4] in subclasses; by definition they must keep the same name and signature. If a virtual function is redeclared in a subclass such that it has the same name but differs in argument type, then the function is considered a *different* function and there is no dynamic link between the two functions. Instead a form of overloading occurs in that two functions with the same name will invoke different code. However the two functions are not both in scope in the subclass; the base class function is hidden by the redeclaration in the subclass. The base class function can be accessed from the subclass by use of the scope resolution operator (::) or via a pointer to the base class which actually points to a subclass object; explicit qualification with the scope resolution operator suppresses the virtual mechanism. A direct result of this lack of overloading in a subclass is that the subtyping relationship between the base class and the subclass is broken since the subclass does not contain the interface of the base class [42, page 6].

In order to maintain dynamic binding of polymorphic types, the signatures of base class functions must not be changed in subclasses (no-variance). Type casts can be used such that base class arguments in the signature of a function can be cast to subclass arguments within the body of a redefined function [42, page 19]. However, this short-circuits the type checking mechanism and the results will be ill-defined if a base class parameter is inadvertently passed to the function instead of a subclass parameter. Apart from the fact that programmer discipline is required for type casting to work, casting from a base class to a derived class is illegal according to The Annotated C++ Reference Manual [40]. Most compilers do not check for this in practice but in theory they should.

In C++ a base class defines an interface specified by the public member functions it declares. Client classes in a client-server relationship[5] can access any features which are in the interface. Subclasses are granted greater access; they have direct access to both the public features and those which are declared as protected but they cannot access the private features directly. C++ also has the notion of *friends*. Classes or individual functions can be defined as friends of a class. Friends can access *any* feature of a class regardless of whether it is private, protected or public. Friends are not inherited by subclasses; each subclass must make its own declarations of friends.

[4] In this report the definitions of the terms *redefinition* and *redeclaration* are the same as those used in [39] which are taken from [43]. Redefinition is defined as changing the implementation of a function while keeping the signature (and specification) the same. Redeclaration is defined as changing the signature, the implementation (and the specification).

[5] A client-server relationship exists when a class (client) declares an attribute of another (server) class, when a client class declares a method with an argument or result of another class or when a method has a local variable of another class.

Eiffel

In Eiffel [44, 45] all method calls are bound dynamically. Those features which are declared as publically exported form the interface to the class; the export status of inherited features can be changed (from public to secret or vice-versa) in a subclass and this may result in the breaking of the subtype relation between the parent class and subclass.

Features may be redefined in Eiffel and furthermore functions may be redefined as attributes in subclasses. Features may also be redeclared within certain limitations. Any change to a type in a signature must be to a conformant type, that is a type may be changed covariantly to a subtype. Preconditions may be weakened but not strengthened and postconditions may be strengthened but not weakened. Attributes may also be redeclared provided that the new type conforms to the old type.

Client classes in a client-server relationship can access any features which are in the interface. Subclasses have direct access to *all* features of the parent class. Selective export is a mechanism which enables a class to export individual features to a particular class or group of classes.

Oberon-2

Oberon-2 [46] is summarised in [47]. Whereas in Eiffel all procedures are bound dynamically, in Oberon-2 (as in C++) the programmer can to some extent decide between static and dynamic binding. Oberon-2 has different forms of procedures: normal, type-bound and variables of procedure types [47, pages 12 – 17]. Normal procedures are statically bound but type-bound procedures and variables of procedure types are dynamically bound.

An Oberon-2 class defines an interface by exporting features either as read-only or as read/write. All users of the class, be they client-servers or subclasses, have to access the class through this interface; subclasses are not granted privileged access. While this ensures that data encapsulation is not broken, it does restrict the way in which type hierarchies can be specialised [48].

Type-bound procedures may be redefined but procedures may not be redeclared in Oberon-2. Syntactic subtyping between super and subclasses is thus ensured but behavioural conformance is not guaranteed since Oberon-2 has no means of enforcing the semantics of methods.

Modula-3

In Modula-3 [49] the term *object type* is used rather than class. Methods are declared in the class interface and are implemented by procedures in the implementation module (in Modula-3 the interface and the implementation must be declared in separate files). The signature of the procedure differs from that of its method in that it has an extra initial parameter which is an instance of the type itself or of an ancestor of the type.

All methods are dynamically bound but static binding can be effected by declaring a procedure, rather than a method, in the interface of the class.

A method may be redefined by assigning, in the implementation module, a different procedure to the method. Redeclaration of a method can be effected by declaring a new method with the same name in the interface of a subclass; the new method need not have the same signature as the superclass method. The redeclaration hides the superclass version, as happens in C++, with the result that the subclass relationship between the superclass and the subclass can be broken.

A class can have more than one interface; the interfaces have different names and the implementation must export both the named interfaces. However, there is no way of restricting the use of an interface to a particular group of classes; in other words there is no equivalent to the friend construct in C++ or to selective export in Eiffel.

A client-server class has access to the features in an interface to a class whereas subclasses have access to all the features in a base class. This is analogous to the situation in Eiffel.

Modula-3 is significantly different from C++, Eiffel and Oberon-2 when the creation of objects is considered. Objects created from a class do *not* all have to have the same behaviour. When an instance of a class is created it is possible to assign *different* procedures to the methods and thereby to change the behaviour from that declared for the class. An object created in this way

is *not* considered to be an instance of the class from which it was created; it is referred to as an *anonymous subtype* of the type of the creating class. In C++, Eiffel and Oberon-2 all the objects created from a given class have the same behaviour which there is no way of altering.

2.4.2 Inheritance and subtyping separated

The languages POOL [2, 28] and Portlandish [29] maintain separate inheritance and subtype hierarchies as described in section 2.3.1 above.

Ada-9X [50] (now Ada-95) is strongly-typed and has a code reuse inheritance hierarchy which is separate from the polymorphic substitutability hierarchy. Programming by extension enables code reuse such that derived types may modify and add to the features of the parent type provided that the parent type is defined as *tagged.* Only record and private types can be tagged. A derived type is *not* in a subtype relationship with its parent type (the types are distinct), direct assignment of a child object to a parent object is not permitted. A derived type has access to parts declared private in a parent type.

A procedure in a parent class which takes a parent class parameter may be redeclared (overridden) in a derived class to take a derived class as parameter but this is *not* covariant redeclaration since there is no subtype relation between the classes. The derived class procedure may call the parent class procedure by *explicitly coercing* the derived class parameter to the parent class parameter. The two procedures with the same name but different parameters are overloaded; which procedure is called is determined by the parameter type and is resolved statically.

Redefining a procedure is not a possibility; for procedures to be resolved by overloading a change in signature is a necessity.

To enable inheritance polymorpism to be effected, Ada-9X introduces the concept of *class-wide* programming. Each tagged type `T` has an associated class wide type `T'Class` which comprises the union of all the types in the tree of derived types rooted at `T`. The values of `T'Class` are the values of `T` and all its derived types and a value of any type derived from `T` can be *implicitly* converted to the type `T'Class`. Direct assignment of child to parent types is possible with class-wide types. Each value of a class-wide type has a tag which identifies its particular type at run time.

In Ada-9X there is therefore an explicit distinction between a specific type such as `Point` and the set or class of types `Point'Class` rooted at the type [51, pages 131 – 133]. Specific types are statically bound and class-wide types are dynamically bound. No such explicit distinction between types is made in C++, Eiffel, Modula-3 or Oberon-2.

2.4.3 Inheritance related to the F-bound

We know of no commercially available language in which inheritance is related to the F-bound relationship. BRUNEL is being developed at Sheffield University.

BRUNEL

In the BRUNEL language [31, 32] inheritance is based on the F-bound relationship which is discussed above in section 2.3.2. Inheritance in BRUNEL is not subtyping rather it is an inclusion relationship among partially-defined type constructors [32, page 34]. Classes are not types instead they are type constructors represented by parameterised polymorphic types [32, page 34]. The class `Point[P]` is a type constructor. When an actual type is generated by legal instantiation of the parameter `P` as in `Point[ColourPoint]`, then all references to `P` in the methods of class `Point[P]` are bound to `ColourPoint`. (The parameterised construct correctly handles the flexible typing of expressions such as: `p: like <anchor>;` in Eiffel.)

It is possible to distinguish between actual types [33, page 56] [32, page 35] such as:

```
p: Point
```

and F-bounded polymorphic types such as:

```
p: Point[ ]
```

Whereas the notation `Point` denotes the *actual* type (that is the most general type generated from the class `Point`), the notation `Point[ ]` denotes an *open-ended* class of `Point` types such that a variable of type `Point[ ]` corresponds to the usual concept of a polymorphic variable in object-oriented programming languages.

Annotations like `Point` which resolve to types can be statically bound by the compiler. Polymorphic annotations like `Point[ ]` may or may not require dynamic binding. "The unification mechanism for resolving the types of polymorphic expressions is well understood from ML" [32, page 40]. In Eiffel, C++, Oberon-2 and Modula-3 the type annotation `Point` may refer to the precise type of object to be stored in the variable or it may refer to a polymorphic class of object-types to be stored there; it is impossible for the compiler to tell which it is. However, Ada-9X is similar to BRUNEL in that the specific type `Point` is distinguishable from the set of class types `Point'Class` and Portlandish has the specific type `PointImpl` and the polymorphic type `Point`.

The use of actual and parameterised types enables distinctions to be made between homogeneous and heterogeneous collections:

```
pointList : List [Point]
```

denotes a homogeneous list of objects of the precise type Point. Whereas

```
pointList : List [Point [P]]
```

denotes a list of polymorphic types constrained by

$\forall$ `t` $\subseteq$ `F-Point[t]`.

Thus it would be possible for `pointList` to contain a (homogeneous) list containing Points or a list containing ColourPoints.

Heterogeneous lists are signified by additional type parameters, thus:

```
pointList : List [Point [S|T]]
```

would denote a polymorphic, heterogeneous list of any type constrained by

$\forall$ `s ,t` $\subseteq$ `F-Point[s, t]`

In other words, a list could contain both Points and ColourPoints.

The type parameters may be replaced at compile time or at run time.

In BRUNEL, the polymorphic abstract types are not only represented by parameterized sets of signatures, they also have axioms to express the semantics of the signatures. The polymorphic type checker ensures signature and assignment conformity at compile time and axiom verification at run time [32, page 22].

BRUNEL supports strict inheritance of type and dependent on this, some lax inheritance of implementation [32, page 30]; the type and implementation hierarchies are not therefore separated and "opportunistic" inheritance of implementation is avoided. In BRUNEL a class has three related semantic interpretations [32, page 30]:

- A *class* is a higher-order type, or type constructor whose type parameter(s) are recursively instantiated in descendent classes to derive new type bounds for inherited methods. Child classes can be regarded as more specific type generators.
- A *type* is generated by replacing all remaining type parameters in a structure. Since type-sharing is viewed in terms of behavioural compatibility, the sharing of (implicitly retyped) methods and axioms is therefore possible and supported.

- A *template* corresponds to the concrete implementation of a type, specifically the table of attributes for each class.

A syntactic class definition in BRUNEL may therefore provide "anything from a completely abstract specification ... to a concrete specification, detailing storage and complete implementation of methods" [32, page 31].

Further comments

The approach taken by C++ to static and dynamic binding forces the programmer to decide in advance how each method is to be bound. Methods are statically bound by default; methods must be declared as *virtual* if they are to be dynamically bound. A static method cannot be redeclared as *virtual* in a sub-class without changing the declaration of the original method in a super-class. It is undesirable to open and alter a class which has been closed. It is also means that existing applications using the old class with static binding now have to use dynamic binding even though they do not need it.

Eiffel adopts dynamic binding by default and has a compiler switch to optimize bindings for a given assembly of classes in a post-processing stage. In BRUNEL the compiler automatically detects the need for static or dynamic binding. Methods which are bound statically in one application may be bound dynamically in another. Within the same application, the same method may be bound statically or dynamically at different call sites.

"In BRUNEL, inheritance is seen firstly as a sharing of behavioural specification and only secondly as a sharing of implementation" [32, page 40]. The requirement that inheritance maintains type consistency restricts the ways in which class implementations can be altered. Simons has found [32, page 40] that inheritance graphs in BRUNEL tend to consist of many small clusters of related types, rather than extended trees. He expects code reuse in BRUNEL to be achieved "more by composition than" by "opportunistic inheritance of implementations" [32, page 40]. Emerald [52] takes an even more extreme view and insists that *all* implementation reuse is through composition; inheritance is reserved for behavioural interfaces.

C++ permits the overloading of function names. For example, '+' can be defined to denote addition (of integers and reals) and concatenation (of strings). BRUNEL does not permit such unrestricted overloading; the re-implementation of an operation must ensure that it has the same, or more restrictive, semantics as the original operation. Two operations with the same name and same semantics must have a common superclass which specifies those semantics [31, page 5].

None of the languages discussed regard classes as a means of "containing" all objects created by the class - the extent is not maintained.

2.5 System development methods

Object-oriented development methods such as those advocated by Booch [53], Coad and Yourdon [54], Wirfs-Brock [55] and Rumbaugh [11] [6] tend not to formalise the semantics of the relationships they use. More recent development methods, Fusion [8] and Syntropy [9], have given some formality to the relationships they use. Our main concern lies with the interpretation of the inheritance relationship in development methods; we have started to investigate this as outlined below.

2.5.1 The influence of the Extended Entity Relationship model on the interpretation of inheritance

The Extended Entity Relationship (EER) model [57] has been used as a basis for the modelling techniques used in development methods such as Fusion and OMT. A brief outline of the EER model is accordingly given here.

[6] Rumbaugh's OMT method is currently being revised and a new method, which is to be combined with that of Booch, is proposed [56].

- An *entity* is defined as a "thing" which can be distinctly identified; for example a person or an event.
- A *relationship* is defined as an association among entities. For example "works-for" is a relationship between a person entity and a company entity.
- Entities in the universe of discourse are classified into different *entity sets* such as `Person` and `Employee`. A predicate associated with each entity set is used to test whether an arbitrary entity belongs to the set. Entity sets are not mutually disjoint; an entity may belong to more than one set.
- A *relationship set* is a mathematical relation among n entities such that its members are tuples representing individual instances of an n-ary relationship.

  $\{[e_1, e_2, ..., e_n] | e_1 \in E_1, e_2 \in E_2, ..., e_n \in E_n\}$
- The *role* of an entity in a relationship expresses the function the entity performs in the relationship. For example, in the relationship set "marriage" defined between entities from the entity set "Person", the first element in a tuple in the set could have the role "husband" and the second element the role "wife".
- *Values* such as "blue", "3", "Ann" are classified into mutually disjoint *value sets* such as "colour", "number", "first name". A predicate is associated with each value set to test set membership. The only property which can be recorded about values is their membership in a value set.
- *Attribute-value pairs* can be associated with an entity or a relationship. An *attribute* is defined as a function that maps from an entity set or a relationship set into a value set or a Cartesian product of value sets.
- An *entity key* is a group of attributes of an entity which can be used to identify the different entities in an entity set. The mapping from the entity set to the corresponding value sets is one-to-one. If an entity cannot be uniquely identified by the value of its own attributes, then relationships can be used together with attributes to obtain the required one-to-one mapping. If there is more than one possible entity key for an entity set, then one is selected as the *entity primary key.*
- Relationships are identified via the entities taking part in the relationship and not via the relationship attributes. The *relationship primary key* is the combination of the primary keys of the associated entities.

Entity sets, relationship sets and value sets can be considered to be the basic types provided by the Entity Relationship model. The model has been extended such that three constructors enable more complex types to be formed:

- *Aggregation* constructs a new entity set out of a relationship set.
- *Generalization* constructs a new entity set from a union of other entity sets. *Specialization* is the reverse process and constructs new entity sets by splitting an entity set into a number of entity sets.
- *Grouping* forms a new entity set from a source set such that each entity consists of a group of entities from the source set.

The new constructors will now be considered in more detail. Particular attention is paid to Generalization/Specialization.

Generalization/Specialization

Generalization creates a new entity set which is *abstract* in the sense that every entity in the new set is a member of one or more of the sets in the union. Formally, if T_1, T_2, ...,T_n are (generalized) entity sets, a *Generalization* is denoted by

$T_1|T_2|...|T_n$

and defines a new entity set T with the meaning that

$t \in T \leftrightarrow \exists T_i(1 \leq i \leq n \wedge t \in T_i)$.

In other words, for every entity in T there exists *at least one* T_i which contains that entity. The entity sets T_i are therefore exhaustive, non-disjoint and are *always* subsets of T. This subset dependency cannot be represented by the unextended ER model.

In a Generalization/Specialization the attributes and the relationships of the "generalized" entity set are *inherited* by the component entity sets. Additional attributes and relationships can be defined for any component set.

The Generalization/Specialization operator results in the formation of entity sets which can be envisaged as creating a hierarchy. An entity may belong to several entity sets in the hierarchy. For example, if the entity sets

```
Person
Employee
Secretary
```

form a hierarchy then an entity existing at the `Secretary` level must also exist at the levels above, that is at the `Employee` level and at the `Person` level. Similarly if an entity is deleted from an entity set then it must cease to exist at any lower level; if an entity is in both the `Employee` and the `Secretary` entity sets then if the entity is deleted from the `Employee` set it must also be deleted from the `Secretary` set.

Aggregation

Whereas in the ER model a relationship set such as "supplier-part" could be defined, aggregation enables a set of relationship sets to be defined such that each entity in the set is itself a *relationship set.* The new entity set can have attributes attached and can take part in relationships. Thus it would be possible to define the set "shipment" with the attribute "date" such that each "supplier-part" relationship set in "shipment" would be associated with a particular date.

(The formal definition does not enforce the existence of a relationship between the entities in an aggregation but an informal comment states that the relationship must exist to make the aggregation meaningful.)

Grouping

New sets are formed from an entity set T such that all the entities in such a set have the same value for an attribute or the same entity in a relationship tuple associated with T. For example, in a set *Employee* a *Grouping* could be indexed on the attribute "salary" such that groups or subsets of *Employee* were formed whereby all the entities in such a group had the same salary. If *Employee* was related to *Company* via the relationship $Works - for$, then $Works - for(company)$ could be used to index groups of employees who all work for the same company.

2.5.2 The Fusion Method

Fusion [8] claims to be a second-generation object-oriented software development method [8, page 1] which integrates and extends earlier methods. Fusion starts with analysis of the problem domain. During analysis two models are built; the object model and the interface model.

"The object model is a modified form of the entity-relationship diagram" [8, page 264]. It is the Extended Entity Relationship model which is being referred to here. An object model is a diagram which has rectangular class boxes (corresponding to entity sets in EER) and relationship

diamonds. A class box represents a named set of objects, it contains the name of a class and may contain attributes. During analysis, attributes may not have type class; that is attribute values may not be objects but are simple values akin to those in EER. Unlike entities, objects have identity. Two objects can have the same values for their attributes without being identical.

Aggregation and relationships are analogous to those in the EER model apart from minor additions - cardinality can be attached to classes in aggregations and there are more elaborate ways of expressing cardinalities in relationships. Since an object can appear as part of many aggregate objects, aggregation is not confined to physical "part-of" relationships. Aggregations in the EER model are used for constructing a new entity set out of a relationship by wrapping up the relationship in an entity. However, in Fusion aggregation is also used to construct a new entity (class) out of other entities (classes); for example the class `Load` is composed of a subset of all `Drums` [8, page 44]. Such a manoeuvre enables the structure of a class to have objects as components; recall that attributes may not have object values in the object model.

Generalization/Specialization is somewhat different in Fusion than in EER and is referred to as *subtyping*. Subtyping is indicated by a triangle symbol. A lower subtype arc connects a (sub)class to the base of a subtype triangle and an upper subtype arc connects the apex of a subtype triangle to a (super)class. As in the EER model, a subclass is a subset of a superclass and each element of a subtype has all the attributes of the supertype. In the EER model the Generalization/Specialization type constructor models non-disjoint, exhaustive subtypes and abstract supertypes. In Fusion, a solid subtype triangle is used to model *disjoint*, exhaustive subtypes which partition the abstract supertype. An outline triangle enables the subtypes to be non-disjoint and also non-exhaustive in which case the supertype may have instances of its own and is not therefore abstract. The EER view of non-disjoint, exhaustive subtypes and abstract supertypes does not have a special triangle and has to be distinguished via the data dictionary as does the disjoint/non-exhaustive case.

All the attributes and relationships of a supertype are "inherited" by the subtypes. A subtype may have additional attributes and relationships but it may not alter the properties of its superclass(es). This strict inheritance rule, which also applies to behaviours, ensures that whatever holds for a supertype object also holds for a subtype object. Consequently, a subtype may always be substituted for a supertype. Since behaviours may not be altered in subtypes, there are severe limitations on the degree of specialization which may be achieved in subtypes. The object model does not address behaviours so inheritance relationships are initially decided by the existence of common attributes and relationships. This changes during the design phase; "the emphasis in design is on behavioural subtyping" [8, page 99]. However, "it is difficult to restrict designs to behavioural subtyping as inheritance is a code reuse mechanism and not a subtyping facility" [8, page 99]. This may refer to the fact that the desired specializations may break the subtyping rules and/or to the fact that not all programming languages enforce the subtyping rules during inheritance.

The overall impression is that generalization/specialization in Fusion is a subset relationship governed primarily by attributes and relationships.

Fusion supports multiple "inheritance" and conversely a class may be a supertype of more than one class as illustrated by the class `TimeStamped` [8, page 264] which is a superclass to the classes `Sample` and `Report`. `TimeStamped` is an example of a class which is designed to provide functionality to other classes rather than to have instances of its own. Such classes, often called *mixins*, do not fit well with the concept of inheritance as a classification relationship; `Report` and `Sample` are not readily regarded as more specialized classes of `TimeStamped`. However, if inheritance is seen as a means of providing common interfaces then `TimeStamped` provides the means of expressing the "is time-stampable" relationship. The Fusion method states that "inheritance should be used for subtyping" [8, page 99].

The interface model describes the externally visible behaviours of the system in terms of an operation model and possibly a life-cycle model. The operation model specifies the behaviour of the system operations declaratively by defining their effect in terms of change of state and the events that are output. The life-cycle model defines the allowable sequences of interactions that a system may participate in over its life-time. Although the interface model analyses the system

operations in terms of the analysed classes, the operations are *not* related to specific classes. Object communication and the attaching of methods to classes is done at the design stage.

During design four models are developed. Object interaction graphs describe how objects interact at run-time to support the system functionality, visibility graphs describe the object communication paths, class descriptions describe the class templates and interfaces and inheritance graphs describe the class inheritance hierarchies. Class inheritance hierarchies are derived from the generalizations (common structures) in the system object model and from the other design models. The object interaction graphs and the class descriptions are used to ascertain common functionality and the visibility graphs are used to identify any more common structure. The following advice is given [8, page 98]: "Each inheritance graph should be rooted in an abstract class that serves only as a definition of an interface. The abstract class should define a minimal attribute set (preferably empty) so that potential subclasses are not restricted in their representation". Such an approach favours the interpretation of subtypes as exhaustive.

It is interesting to note that class descriptions use the keyword "is-a" to describe inheritance; for example, `class StoreBuilding is-a Building` [8, page 91]. The meaning of is-a is that "the set denoted by a class description is a subset of all the sets referred to in `is-a` clauses" [8, page 289]. Information has been lost in that "is-a" does not convey the fact that StoreBuilding was analysed as being a *disjoint* subtype of the *abstract* type Building.

In Fusion a class box represents a named set of objects [8, page 266]. According to Civello [58, page 384] such an extensive view of the meaning of classes leads to the temptation to treat a class as a collection of all its instances which Civello regards as bad practice. He argues that at the conceptual level such a view confuses the member-collection relation, based on the extrinsic properties of a group of objects, with the class membership relation which is based on the intrinsic properties of the individual class members. Practically, it will be necessary for a class to have to both a) define and create instances of a class and b) to keep and manage all the instances of a class. Although Fusion adopts the notion of *collection* classes during design to represent a collection of objects (such as files), collections are not treated as first class citizens. When a message is sent to/from a collection, the message is deemed to have been sent to/from each object in the collection rather than to the collection as a whole.

2.5.3 The Syntropy Method

Syntropy does not claim to be a software development method, rather it offers a range of techniques which developers should adopt as they see fit. Diagrammatic notation is used but the semantics of diagrams are given formally. Two of the diagrammatic notations used are OMT and Harel statecharts as used by Rumbaugh [11], another is mechanisms from Booch [53]. Z [59] is used for the formal notation.

Three models are built; the essential model is based on objects and events, the specification model states what the software will do and the implementation model describes the objects in the executing software and how they will communicate.

The Essential Model - static view

In the essential model, the term *object type* is used to describe the structure of a model of a situation in the world, rather than the term class, on the grounds that class is too closely linked to object-oriented programming languages whereas Cook and Daniels are concerned with object capabilities in the world [9, page 31]. The properties of object types are not distinguished as data and operations; rather properties are regarded as "observable" - how they are observed is decided when the software is designed. It is important to stress that an object model in Syntropy is not a model of stored data.

Value-typed properties, known as properties, can be considered to be functions that return value-typed results. Properties which have object-types are called associations. An association between two object types A and B describes two functions, one mapping from A to B and the other mapping from B to A [9, page 34]. (In EER and Fusion, the relationships which describe

associations are regarded as sets of tuples.) Associations can have properties and can be related to other object types as in OMT; in EER and Fusion relationships cannot be related to another entity unless the aggregation construct is used. In Syntropy object types that represent a binary association can be used in the same way as the aggregation construct is used to wrap up relationships in Fusion. However, the Syntropy view of aggregation is very different to that of Fusion. In Syntropy, aggregations (whole-part relationships) are distinguished from associations by the fact that in an aggregation the life-times of the 'parts' are contained within the life-time of the 'whole'. The parts are permanently attached to the whole and cannot be removed from the whole without being destroyed; if the whole is destroyed then so are the parts [9, page 39].

Value-typed properties belong to a single object which manages the states of the properties. Associations between two objects belong to both the objects and can be managed by either object [9, page 107].

Apart from describing properties and associations, object types also describe the set of objects conforming to the type [9, page 83]. An object type `Person` could have a value-property *name* and association *employer* [9, page 82] and it would also represent the set of all persons in the system.

Type Extension - Essential Model

Types may be extended by defining subtypes. A subtype 'inherits' all the properties, constraints and associations of its supertype. Cook and Daniels do not mean inherit in the sense that it is used in object-oriented programming. They define subtyping to imply object conformance; an object which conforms to a subtype must also conform to the supertype. "The exact rules governing conformance vary between the essential, specification and implementation models but, broadly, the subtype can extend the capabilities of the supertype but not restrict them" [9, page 43].

Type extension is indicated by an open triangle such that the apex is joined to the supertype and a line along the base joins the subtypes. The subtypes are disjoint but not exhaustive. Exhaustive disjoint subtypes are represented by the type constraint *abstract* in the supertype. Non-disjoint subtypes are represented (somewhat illogically) by connecting the subtypes to the supertype with *separate* triangles.

The semantics of the models are such that objects cannot change the set of types to which they conform in their life-times. Thus if the type `Person` has subtypes `UnemployedPerson` and `EmployedPerson`, then a person who is unemployed can never be employed [9, page 52]. To overcome this problem, Cook and Daniels introduce *state types* which are represented by a diagonal line across the top left corner. Objects of state types cannot be created as conformance to state types changes as an object (of the super-type) changes type. However, the use of state types enables the properties and associations gained and lost as a dynamic object moves from one state to another to be defined.

Multiple supertypes are possible and a type with multiple supertypes 'inherits' the union of the properties, constraints and associations of its supertypes. (There must not be any name clashes between the supertypes.)

Subtypes may override (redefine) the features of a supertype in the essential model provided that structural conformance is maintained. The definition given by the subtype replaces and hides the definition given by the supertype. Structural conformance only considers navigable structure and properties. "If navigation of a model yields a set of objects of a particular type, then observing the properties of the objects in the set should not produce any surprises, even if some (or all) of the objects also conform to a subtype. The kind of surprise we have in mind would be a constraint violation or a property of the wrong type" [9, page 70]. Associations may be overridden in order to change the multiplicity constraints such that the subtype range is within the supertype range; subtypes must not be allowed to have ranges not permitted by supertypes. Associations may also be overridden to change both the associated types to subtypes or to change the aggregation constraint.

Properties may be overridden. The type of a property parameter must not be changed. Type invariants may be strengthened but not weakened; that is the subtype constraint must imply the supertype constraint.

These overriding rules are consistent with the subtyping rules of Cardelli and Wegner [24] and with the axiomatic rules of Simons [26]. However, the type, T, of a property may be changed to a subtype of T which may result in the subtyping rules being broken.

The Essential Model - dynamic view

Type views model static information; the dynamic behaviour of the essential model is modelled by *events*. Events, which can only occur in particular sequences, cause changes in state. The changes in state which are effected by the possible ordering of events are modelled by state machines which are also used to model dynamically acquired properties and associations [9, page 91]. State machines describe the change in state of specific object types, Cook and Daniels believe [9, page 90] that behaviour should be attached to objects - they do not like the approach which shows event schemata defined in terms of the model as a whole (as used in Fusion) rather than in terms of individual objects.

The essential model must have a type, the *initial type*, which only has one instance, the *initial object*. All the objects that make up an instance of a model must be reachable by navigation from the initial object [9, page 81]. The motivation behind initial objects is that there is always an object in which behaviour can be expressed [9, page 119].

The Specification Model

The specification model specifies the software in terms of the effects of incoming events on state and of any outgoing events. One of the main purposes of the specification model is the allocation of system behaviour among individual objects [9, page 148].

Object types and state charts are used but the notation for state charts is extended and the interpretation of the notation is changed slightly. For example, in the essential model the failure of a pre-condition means that an event cannot happen, in the specification model such a failure means that the software's response to the event is undefined. The structure of state charts is inherited by subtypes.

The Implementation Model

The implementation model defines the flow of control in the software. In the essential and specification models events are instantaneous and broadcast; they may be detected simultaneously by many objects of many types. Cook and Daniels consider that in the essential model this reflects the real world and in the specification model avoids over-specification [9, page 259]. In the implementation model the message interactions between objects are designed because these are needed for software construction. *Mechanisms* are used to show the sequence of messages which are sent when an object receives a particular message. Mechanisms use the basic syntax of of object views, the numbering scheme used for messages in mechanisms was adopted from Fusion's object interaction graphs [9, page 192]. Messages invoke operations as in object-oriented programming languages.

The implementation model does not address the way in which value-properties are implemented or modified. However, properties in the specification type view are replaced by *observers* and *updaters*. An observer operation does not change the state of an object, it merely obtains information for the sender of a message. An updater operation can change the state of the object executing it and, indirectly, the state of other objects. Updaters reflect events in the specification model. If an object's associations are to be made visible to the object's clients, then an observer operation must be added to the type view of the object.

The rules for structural conformance between a type and its subtypes apply to the implementation model [9, page 169] - an object which sends a message to a supertype can send the same message to a subtype without being 'surprised' at the result. The rules are:

- The subtype must provide the same observers as the supertype, or a superset thereof.

- The subtype must provide the same updaters as the supertype, or a superset thereof.
- The return type of a subtype operation must be the same as, or a subtype of, the return type of the supertype operation (co-variant result types).
- The parameter types of a subtype operation should be the same as the parameter types of the supertype operation (no-variance of parameter types).

Cook and Daniels admit that the last rule is over restrictive and that contra-variant parameter types could have been allowed. They claim that an advantage of no-variance is that it makes it clear when an operation is being overridden. (It also agrees with the policy in C++.)

The dynamic state behaviour of every object type in an implementation model can be described using a separate state chart. If all an object's operations are valid at all times, a state chart is not needed.

Subtypes and conformance

"Sub-typing is a relationship of substitutability with respect to a given set of expectations" [9, page 194]. Cook and Daniels remark that the expectations are different for different models. They acknowledge [9, page 194] that there are loopholes in their modelling notations which can allow type conformance to be broken.

Structural type conformance is based on the type view and is almost the same for the three models used in Syntropy. Behavioural conformance, based on the statecharts, differs between the models.

In the specification model, substitutability is based on *event detection*; subtypes conform to supertypes in the way that they respond to events. If a supertype responds to a particular sequence of events and ends up in a particular state, then a subtype should respond to the same sequence of events and end up in the same state. Complete substitutability in a system is not required, only substitutability as an event receiver. Events generated by subtypes do not have to correspond to events generated by supertypes and subtypes do not have to fail to respond to event-sequences that supertypes would fail to respond to. Consequently, whereas the failure of a pre-condition in the essential model implies that the event cannot happen, such a failure in the specification model implies that the response to the event is undefined.

There are thirteen detailed rules for specification type conformance which will usually ensure event detection conformance. Three examples of situations which will break the rules are given [9, page 212]; the most common occurs when a property which has been restricted in the subtype can be updated such that it is in range for the supertype but not for the subtype. The strategy for overcoming this situation involves a *requestUpdate* event which is similar in purpose to the request operation we had to adopt when trying to effect subtyping in CCS, see section 2.6.1 below.

In the implementation model, substitutability is based on the receiving of messages. The rules for structural and behavioural conformance are the same as for the specification model except that the rules are interpreted in terms of message reception rather than event detection. Type-conformance is not concerned with the messages sent by instances of a type. Cook and Daniels point out [9, page 215] that with their definition of conformance all sequences of messages accepted by a supertype will be accepted by a subtype and will leave it in a corresponding state. This is a much stronger requirement than that defined by many object-oriented programming languages (including those we have discussed above) which require that an operation existing in a supertype exists in a subtype but which say nothing about operation *sequences*. If a subtype is unable to respond to all the message sequences of the supertype, programs will not be robust and could fail.

Essential model type conformance is simply structural and the rules are effectively the same as for the specification model and the implementation model. Behavioural type conformance is not considered so event sequences do not have to be compatible.

2.5.4 The OMT, Coad and Yourdon and RDA Methods

As yet we have only had a rather superficial look at these methods; however, a comparison of the three approaches can be found in Mayes [38]. The inheritance relationship is similar in all three methods [38, page 22], in particular all methods recommend that subclasses inherit all the data and operations of their superclasses. The methods also advocate the use of an abstract superclass for a function such as `display` which is used by more than one class. As Mayes points out, such a practice results in increased multiple inheritance. Once again we see inheritance being used both as some sort of classification mechanism and also as a means of providing common interfaces.

OMT

In OMT the distinction is made between disjoint (represented by a hollow triangle) and non-disjoint (represented by a solid triangle) subclasses and between abstract and concrete superclasses. Rumbaugh [56, page 25] states that subclasses tend to be disjoint, but that they can overlap.

In OMT Rumbaugh et al. [11, page 39] describe generalization as "the relationship between a class and one or more refined versions of it". They note that "generalization is sometimes called the 'is-a' relationship because each instance of a subclass is an instance of the superclass as well" and consequently "any operation on any ancestor class can be applied to an instance of a subclass". Subclasses may override a superclass feature but the signature must not be overridden [11, page 42]. However, Rumbaugh et al. [11, page 42] go on to say that "tightening the type of an attribute or operation argument to be a subclass of the original type ... must be done with care" which seems to be suggesting that the signature *can* be altered and that covariant redefinition of an argument type requires caution. This is confirmed when they say [11, page 64] that "tightening the types of arguments ... may be necessary to keep the inherited operation closed within the subclass". The example they give has `Set` with the operation `add(object)` as the superclass and `IntegerSet` with the more restricted operation `add(integer)` as the subclass. However, by allowing `IntegerSet` to be a subclass of `Set` they have nullified their claim that "any operation on any ancestor class can be applied to an instance of a subclass" since an `add(object)` may be applied to a `Set` object but *not* to an `IntegerSet` object. It would appear that they are confusing the is-a relation and the subtype relation. As LaLonde and Pugh [12, pages 58 –59] point out, `IntegerSet` is not a subtype of `Set` since an `IntegerSet` object cannot be substituted for a `Set` object in case an attempt is made to add an object (other than an integer) to the `IntegerSet` object. However, an `IntegerSet` is a specialized form of `Set` in which the elements are restricted to integers. "The members of the 'Set of Object' include as a special case those objects that are called 'Set of Integer' " [12, pages 58 –59]; hence an `IntegerSet` is in an is-a relationship with a `Set`. The conflict between specialization (is-a) and subtyping is once again apparent.

Coad and Yourdon

Coad and Yourdon [10] use an open semi-circle to denote generalization-specialization (an open triangle is used for aggregation). No explicit distinction is made between joint and disjoint subclasses but disjoint subclasses can be illustrated either by the use of redundant information in more than one subclass [10, page 87] or by multiple inheritance [10, page 88]. Abstract superclasses which have no objects (the subclasses are exhaustive) are distinguished from superclasses having objects (the subclasses are non-exhaustive). Coad and Yourdon stress [10, page 86] that generalization-specialization structures should reflect generalization-specialization in the problem domain and should not be used "for the sake of extracting out a common attribute". Although attributes are used as the basis for developing generalization-specialization structures in the early stages of analysis, Coad and Yourdon claim [10, page 144] that both services (behaviour) and attributes are important and that they tend to work back and forth between the two.

Wirfs-Brock

Wirfs-Brock bases inheritance hierarchies on shared behaviour rather than shared structure [55, page 187]. Much emphasis is placed on the development of abstract classes - "the goal is to find as many abstract superclasses as possible" [55, page 47]. This factoring out of common behaviour is aimed at providing maximum code reuse when new functionality is added to an application [55, page 188].

2.6 The formal specification of inheritance

There has been much interest in the application of object-oriented mechanisms to formal specification methods. A review of the current state of research, given in [60], makes the point [60, page 15] that not all the methods have well defined formal semantics.

Here we summarise some early work we undertook concerning the representation of inheritance in formal languages.

2.6.1 CCS

The motivation for considering CCS [61] for object-oriented specification arose from the desire to be able to express the time-ordering of operations, the belief that the language might be able to express the heterarchical interaction between active objects and in particular we wanted to ascertain whether CCS could be used to model inheritance. It is interesting to note that Cook and Daniels acknowledge [9, page 151] that their treatment of events in essential and specification models in Syntropy arose partly from CSP (Communicating Sequential Processes).

The results of our attempt to use CCS to specify inheritance are summarised in the Euromicro paper by Buchanan [62] and given in more detail in our technical report [63]. We interpreted a CCS agent expression as an interface to a class and treated inheritance as a subtype relationship such that a subtype must be able to effect the same sequences of operations as its supertype. CCS is useful for expressing any necessary sequences of operations but we did not have much success in our attempt to use it for subtype inheritance. We were able to express subtype inheritance by adding extra complexity to our specification but with the even more serious disadvantage of having non-deterministic behaviour in the subtype. While the subtype could satisfy the sequence of operations required of the supertype, the non-determinism meant that it might not be able to effect the extra operation required of the subtype. In order to overcome the non-determinism we had to modify the derived type by adding a 'requestOperation' operation. This resulted in the subtype relationship being broken due to the fact that the 'requestOperation' was needed not for the new operation in the subtype but for one of the inherited supertype operations. The subtype relationship could be restored by adding the 'requestOperation' to the supertype as well, thereby redefining the original supertype.

We came to the conclusion that in order to model subtype inheritance with a suitable degree of clarity we need either to find another way of using CCS or we should use another calculus.

2.6.2 OBJ and FOOPS

We undertook a small experiment to ascertain the viability of representing inheritance in OBJ3 [64], a functional programming language based on order-sorted algebra. The experiment is described in a report by Buchanan [65, pages 84–96] and is summarised in the short paper [66] we presented at Euromicro.

The ability to declare one sort (the OBJ3 term for type) as a subsort of another makes it possible to represent inheritance polymorphism in the sense that a subsort can be used in place of a supersort. We were able to define a sort `TimeDiary` as a subsort of `Time` such that operations defined for values of sort `Time` were able to accept values of sort `TimeDiary`. The operations defined in the TIME module were 'inherited' , actually imported, by the TIMEDIARY module, extra operations could be defined in the TIMEDIARY module using either `TimeDiary` or `Time` sorts and operations initially defined in the TIME module could be redefined in the TIMEDIARY module.

Since there is no concept of pre- and post-conditions in OBJ3 it is possible, but inadvisable, to alter the semantics of a redefined operation.

In another experiment we used the `Card` example described in [63]. The results of this experiment, reported at a staff seminar on 6 December 1991, reinforced those obtained for the `Time` experiment. In addition they showed that when an operation is redeclared overloading occurs such that both the new and the old versions of the operation are in scope in the module in which the redeclaration occurs. The fact that a subsort, `CardPlus`, could be used where a supersort, `Card`, was expected had the unexpected side effect that it was possible to construct a `CardPlus` with more than one PIN number which had not been the intended semantics for `CardPlus`. Further research is needed to establish the ramifications of this result.

The inheritance we modelled using OBJ3 was subsort inheritance. Sets of values can be defined as subsets of other values but the values themselves are immutable; a sort Time can have a subsort TimeDiary but values of either sort are constants. OBJ3 has no concept of state so mutable entities are not naturally represented and object inheritance is not supported. However, importation in OBJ3 supports module 'inheritance' which enables code reuse.

FOOPS [67, 68] is an object-oriented programming language based on OBJ3. The type system of FOOPS distinguishes between sorts (collections of values), classes (collections of objects) and modules in which several related sorts and classes may be declared together. Inheritance of sorts, classes and modules is supported. Sort and class inheritance are used to represent hierarchical classifications. Module inheritance is used for code reuse via importation; in this respect it is similar to private inheritance in C++.

An interesting feature of FOOPS is that it allows co-variant redefinition of methods. This is said to be because it captures "the more common situation in practice" [68, page 48] and because it obeys the algebraic semantics of FOOPS. A mechanism involving retracts is used such that fatal run-time errors are avoided if the object is dynamically of the 'wrong' class [68, page 48].

2.6.3 Z and Object-Z

Z [69, 59] can be used to represent state and operations and although these are not tied together as in classes and objects, it has been argued [70, page 4] that Z can, albeit with some difficulty, be regarded as object-based. State can be 'inherited' via schema inclusion and operations can be 'inherited' via promotion (see for example [71, page 103] and [70, page 12]). In order to promote an operation, a general updating schema is defined such that a particular instance (such as a file or a quadrilateral) in a system can be changed in an as yet unspecified way. An operation for an individual file or quadrilateral can then be promoted (combined with the general schema) such that it works on one particular instance in a system containing many instances. However, inheritance of state and operations together is not supported and hence Z is not object-oriented.

Various approaches to applying object-oriented structuring to Z are reviewed in [70]. One of these is Object-Z [72, 70]. Gordon Rose points out [70, page 59] that a major problem with large Z specifications is that "inferring which operation schemas may affect a particular state schema requires examining the signatures of all operation schemas". Object-Z overcomes this problem by confining individual operations to refer to a particular state schema in class definitions. Class definitions can be related by inheritance and by object references and a class is used to represent the entire system.

A class can have a visibility list which restricts access to any listed features. Visibility is not inherited which aids reusability but could cause problems if a subtype relationship was to be maintained.

Z has a formal semantics but until very recently Object-Z did not; work aimed at providing a formal semantics is given in [73] and Lano [74] references the new work by G.Smith on an axiomatic semantics and reasoning system. Object-Z adopts object reference semantics, rather than object semantics. A significant advantage of this is that when aggregates are modelled, operations on individual members of the aggregates can be achieved without the need for framing and promotion [70, page 63].

Another advantage of Object-Z is that it enables history invariants to be used to constrain the time-ordering of operations on objects.

As yet we have not undertaken any experimental work with Object-Z or any of the other object-oriented Z notations.

3 Conclusions

Inheritance enables code to be reused in a manner which is more flexible (due to the 'self' mechanism) than that achieved by composition. Inheritance may also enable subtype (substitutability) and is-a (specialization) relationships to be expressed but the implementation of inheritance in many programming languages makes this problematical. If inheritance is allied to subtyping the ability to represent the specialization required of is-a may be severely restricted by the contravariance rule. If inheritance is separated from subtyping then its use to represent is-a specialization will depend on the disciplined reuse of code. If inheritance is to be the means by which specialization is-a relationships are maintained, then one way of achieving this is via the F-bound relationship.

The is-a and inheritance relationships are often considered together, particularly in object-oriented modelling, but they do not have to be. Is-a can also be represented by repetition of code or it could be inferred by the compiler. Inheritance can be used to maintain heterogeneous collections or to reduce the need for case statements. The fact that there are so many interpretations of the is-a relationship adds to the confusion.

The subtype relationship is governed by substitutability; a supertype should always be able to be replaced by a subtype. Subtyping thus enables polymorphic functions to be used safely. Substitutability requires contravariance and thereby restricts the ability of the subtype relationship to represent the specialization sometimes required for the is-a relationship.

The F-bound relationship enables type parameterised functions to be inherited such that recursive references are bound to the new types created by inheritance. The types created by inheritance are not necessarily in a subtype relationship but the F-bounded type functions from which the types are created must be in a subtype relationship. Polymorphic functions can be quantified over F-bounds and can thus be applied to any type within the quantifying F-bound.

4 Future work

The main objective of the research is to ascertain the effects on object-oriented development of the introduction into object-oriented programming languages of F-bounded type systems to represent the semantics of inheritance.

4.1 F-bounded object-oriented programming languages

An investigation into the nature of object-oriented programming languages which are based on F-bounded type systems will be undertaken. The aim will be to to determine how many degrees of freedom exist within F-bounded constraints. The various ways in which polymorphism can be represented will be addressed.

The languages Brunel and Rapide will be used as exemplars of languages adopting F-bounded type systems. The F-bound is central to the inheritance relationship in Brunel but appears to be very much less so in Rapide.

We shall seek to find out what can and cannot be represented with an F-bound model. If capabilities which are usually represented by inheritance are 'lost', we shall need to ascertain whether it really matters if they are represented by inheritance and if not, whether they can be provided by other means.

4.2 Application of F-bounded relationships during analysis and design

Design patterns [75] will be investigated to ascertain any implications of the F-bounded relationship for such patterns.

Given that the F-bound is found to affect design patterns, the wider implications for design and analysis will be considered. Small case studies (such as the 2D/3D Point in [42]) and larger case studies from the literature (such as the ECO Storage Depot [8], the Automated Teller Machine [11] and the bottling plant [9]) will be used.

The effect of the F-bounded inheritance relationship on the subtyping rules of Syntropy will be examined.

References

[1] Peter Wegner. The Object-Oriented Classification Paradigm. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming, 2nd edition.* The MIT Press, 1988.

[2] Pierre America. Inheritance and Subtyping in a Parallel Object-Oriented Language. In *Proceedings of the European Conference on Object Oriented Programming*, pages 234–242, 1987.

[3] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is Not Subtyping. *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, January 1990.

[4] James M. Armstrong and Richard J. Mitchell. Uses and abuses of inheritance. *Software Engineering Journal*, pages 19 – 26, January 1994.

[5] Russel Winder. *Developing C++ Software.* John Wiley and Sons Ltd., Chichester, West Sussex, 1991.

[6] Barbara Liskov and Jeanette M. Wing. A New Definition of the Subtype Relation. In Oscar M. Nierstrasz, editor, *ECOOP '93 - Object-Oriented Programming*, pages 118–141. Springer-Verlag, 1993.

[7] Richard Hull and Roger King. Semantic Database Modelling: Survey, Applications and Research Issues. *ACM Computing Surveys*, 19(3):201 – 260, September 1987.

[8] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, and Paul Jeremes. *Object-Oriented Development THE FUSION METHOD.* Prentice Hall Inc, Englewood Cliffs, New Jersey 07632, 1994.

[9] Steve Cook and John Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy.* Prentice Hall, Hemel Hempstead, UK, 1994.

[10] P. Coad and E. Yourdon. *Object-Oriented Analysis.* Prentice Hall, Inc, Englewood Cliffs, New Jersey, second edition, 1991.

[11] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall International, Inc., Englewood Cliffs, New Jersey 07632, 1991.

[12] Wilf LaLonde and John Pugh. Subclassing $\neq$ subtyping $\neq$ Is-a. *Journal of Object-Oriented Programming*, pages 57 – 62, January 1991.

[13] Peter Canning, William Cook, Walter Hill, and Walter Olthoff. F-Bounded Polymorphism for Object-Oriented Programming. *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, 1989.

[14] R.G.G. Cattell. *Object Data Management Object-Oriented and Extended Relational Database Systems.* Addison-Wesley, Wokingham, England, revised edition, 1994.

[15] Scott Danforth and Chris Tomlinson. Type Theories and Object-Oriented Programming. *ACM Computing Surveys*, 20(1), March 1988.

[16] James M. Armstrong. *The roles of inheritance in software development.* PhD thesis, Information Technology Research Institute, Brighton Polytechnic, 1991.

[17] William Cook and Jens Palsberg. A Denotational Semantics of Inheritance and its Correctness. *OOPSLA '89 Proceedings*, 1(6), October 1989.

[18] Peter Wegner. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger* , 1(1), 1990.

[19] Geir Magne Hoydalsvik and Guttorm Sindre. On the purpose of Object-Oriented Analysis. *OOPSLA*, pages 240 – 255, 1993.

[20] Ronald J. Brachman. What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks. *I.E.E.E Computer*, 16:30 – 36, October 1983.

[21] Ronald J. Brachman. "I lied about the Trees" Or, Defaults and Definitions in Knowledge Representation. *The AI Magazine*, pages 80 – 93, Fall 1985.

[22] K.M.Buchanan and R.G.Dickerson. An Investigation of Types leading to an Examination of some aspects of F-bounded Interfaces and the Type Classes of Haskell. Technical Report 154, Division of Computer Science, University of Hertfordshire, College Lane, Hatfield, Herts AL10 9AB, 1993.

[23] James Rumbaugh. Disinherited! Examples of misuse of inheritance. *Journal of Object-Oriented Programming*, 5(3):22 – 24, February 1993.

[24] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4), December 1985.

[25] C.Strachey. Fundamental Concepts of Programming Languages. Research Report, Oxford University Programming Research Group, 1967.

[26] A.J.H.Simons and A.J.Cowling. A Proposal for Harmonising Types, Inheritance and Polymorphism for Object-Oriented Programming. Research Report CS-92-13, The University of Sheffield, Department of Computer Science, Regent Court, University of Sheffield, 211 Portobello St., Sheffield S1 4DP, 1992.

[27] W.R.Cook. A Proposal for Making Eiffel Type-safe. *The Computer Journal*, 32(4), 1989.

[28] Pierre America and Frank van der Linden. A Parallel Object-Oriented Language with Inheritance and Subtyping. *Proceedings of the European Conference on Object Oriented Programming/OOPSLA '90 Proceedings*, 21(25):161 – 168, October 1990.

[29] Harry H. Porter III. Separating the subtype hierarchy from the inheritance of implementation. *Journal of Object-Oriented Programming*, 4(9):20 – 29, February 1992.

[30] A.J.H.Simons and Low E.K. and Ng Y.M. An Optimizing Delivery System for Object-Oriented Software. Research Report CS-93-18, The University of Sheffield, Department of Computer Science, Regent Court, University of Sheffield, 211 Portobello St., Sheffield S1 4DP, 1993.

[31] A.J.H.Simons. BRUNEL: A Strongly-Typed, Portable Object-Oriented Language and Programming Environment. Research Report CS-91-07, The University of Sheffield, Department of Computer Science, Regent Court, University of Sheffield, 211 Portobello St., Sheffield S1 4DP, 1991.

[32] Anthony J. H. Simons, Low Eng Kwang, and N G Yee Mei. An optimizing delivery system for object-oriented software . *Object Oriented Systems* , 1(1):21 – 44, September 1994.

[33] A.J.H.Simons. Introduction to object-oriented type theory. In *ACM Conference on OOPSLA-93 Tutorial Notes*, 1993. 62 pp.

[34] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *Proceedings 16th ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.

[35] Tsvi Bar-David. Practical consequences of formal definitions of inheritance. *Journal of Object-Oriented Programming*, pages 43 – 49, July/August 1992.

[36] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, Hemel Hempstead, 1988.

[37] Richard Mitchell, John Howse, Stuart Kent, and Ian Maung. Software contracts: a way forward. Position paper for ICSE-17 Workshop on Research Issues in the Intersection of Software Engineering and Programming Languages, 1994.

[38] Jeanne Audrey Mayes. *A Technique for Clarifying the Implementation of Relationships between Objects to Enhance Software Reuse.* PhD thesis, University of Hertfordshire, 1995.

[39] Audrey Mayes and Mary Buchanan. A Comparison of Eiffel, C++ and Oberon-2 . Technical report 191, The University of Hertfordshire, College Lane, Hatfield, Herts AL10 9AB, 1994.

[40] Margaret A.Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, Massachusetts, second edition, 1990.

[41] Stanley B. Lippman. *C++ Primer.* Addison-Wesley, Reading, Massachusetts, 1991.

[42] Mary Buchanan. Overloading and Polymorphism in the interpretation of Inheritance in C++. Technical Report 202, Division of Computer Science, University of Hertfordshire, College Lane, Hatfield, Herts AL10 9AB, 1994.

[43] Laszlo Boszormenyi. A Comparison of Modula-3 and Oberon-2. *Structured Programming*, 14, 1993.

[44] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall International (UK) Ltd , 66 Wood Lane End, Hemel Hempstead, Hertfordshire, HP2 4RG, 1988.

[45] B. Meyer. *Eiffel: The Language.* Prentice Hall, Hemel Hempstead, 1992.

[46] Martin Reiser and Nicholas Wirth. *Programming in Oberon.* Addison-Wesley Publishing Company, New York, 1992.

[47] Audrey Mayes and Mary Buchanan. The Oberon-2 Language and Environment. Technical Report 190, The University of Hertfordshire, College Lane, Hatfield, Herts AL10 9AB, 1994.

[48] Libero Nigro. On the type extensions of Oberon-2. *ACM SIGPLAN Notices*, 28(2), 1993.

[49] Samuel P. Harbison. *Modula-3.* Prentice Hall, Englewood Cliffs, New Jersey 07632, 1992.

[50] Office of the Under Secretary of Defense for Acquisition, Washington, DC 20301. *Introducing Ada 9X*, Third edition, September 1993.

[51] S. Tucker Taft. Ada 9X: From Abstraction-Oriented to Object-Oriented . *OOPSLA '93*, pages 127 – 136, 1993.

[52] Rajendra K. Raj and Ewan Tempero and Henry M. Levy and Andrew P. Black and Norman C. Hutchinson and Eric Jul. Emerald: A General-Purpose Programming Language. *Software-Practice and Experience*, 21(1):91 – 118, January 1991.

[53] Grady Booch. *Object-Oriented Design With Applications.* Benjamin/Cummings, London, 1991.

[54] Peter Coad and Edward Yourdon. *Object-Oriented Analysis.* Prentice-Hall International, London, 2nd edition, 1991.

[55] R.Wirfs-Brock and B.Wilkerson and L.Weiner. *Designing Object-Oriented Software.* Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[56] James Rumbaugh. OMT: The object model. *Journal of Object-Oriented Programming*, pages 21 – 27, January 1995.

[57] Antonio L. Furtado and Erich J. Neuhold. *Formal Techniques for Data Base Design.* Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1986.

[58] Franco Civello. Roles for composite objects in object-oriented analysis and design. *OOPSLA '93 Proceedings*, pages 376–393, 1993.

[59] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z.* Prentice Hall International (UK) Ltd, 66 Wood Lane End, Hemel Hempstead, Hertfordshire HP2 4RG, 1991.

[60] Antonio Ruiz-Delgado and David Pitt and Colin Smythe. A Review of Object Oriented Approaches in Formal Specification. Reviewed in March 1995 for possible publication, 1995.

[61] R.Milner. *Communication and Concurrency.* Prentice Hall International (UK) Ltd, 66 Wood Lane End, Hemel Hempstead, Herts HP2 4RG, 1989.

[62] Mary Buchanan, Bob Dickerson, Carol Britton, and Martin Loomes. Using CCS for the Specification of Inheritance. *Microprocessing and Microprogramming*, 39:93 – 96, 1993.

[63] M.Buchanan and R.G.Dickerson. CCS and Object-Oriented Concepts. Technical Report 140, Division of Computer Science, University of Hertfordshire, College Lane, Hatfield, Herts AL10 9AB, 1992.

[64] Joseph A. Goguen and Timothy Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, August 1988.

[65] Mary Buchanan. The Phantom of the Object. MSc report, School of Information Sciences, Hatfield Polytechnic, College Lane, Hatfield, Herts AL10 9AB, 1990.

[66] Mary Buchanan and Carol Britton. Formal Specification and Object-Oriented Design. *Microprocessing and Microprogramming*, 34:19 – 21, 1992.

[67] Joseph A. Goguen and Jose Meseguer. Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming, 2nd edition*, pages 417 –477. The MIT Press, 1988.

[68] Adolfo J. Socorro Ramos. *Design, Implementation and Evaluation of a Declarative Object-Oriented Programming Language.* PhD thesis, Oxford University Computing Laboratory Programming Research Group , 1993.

[69] J.M.Spivey. *The Z Notation, A Reference Manual.* Prentice Hall International (UK) Ltd, 66 Wood Lane End, Hemel Hempstead, Hertfordshire, H P 2 4 R G, 1989.

[70] Susan Stepney and Rosalind Barden and David Cooper, editor. *Object Orientation in Z.* Springer-Verlag, Berlin Heidelberg New York, 1992.

[71] Ian Hayes, editor. *Specification Case Studies.* Prentice-Hall International (UK) Ltd, 66 Wood Lane End , Hemel Hempstead, Hertfordshire, HP2 4RJ, 1987.

[72] David A. Carrington and David Duke and Roger Duke and Paul King and Gordon A. Rose and Graeme Smith. Object-Z: An object-oriented extension to Z. In S. Vuong, editor, *Formal Description Techniques II, FORTE '89*, pages 281–296. North Holland, 1990.

[73] David Duke and Roger Duke . Towards a Semantics for Object-Z. In Dines Bjorner and C.A.R.Hoare and H.Langmaack, editor, *Formal Methods in Software Development, Kiel, volume 428 of Lecture Notes in Computer Science*, pages 244–261. Springer Verlag, 1990.

[74] K.Lano. Formal Methods and Object-orientation: A Historical Perspective. *FACS Europe*, 2(1):13 – 16, 1995.

[75] Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Massachusetts 01867, 1995.