AN OBJECT DATA MODEL WITH ROLES [1]

A. Albano, R. Bergamini, G. Ghelli, R. Orsini

Dipartimento di Informatica – Università di Pisa
Corso Italia 40, 56100 Pisa – Italy

Abstract

Fibonacci is a strongly typed, object-oriented database programming language with a new mechanism to model objects with roles. Traditional object-oriented programming languages do not have the possibility of changing dynamically the type of an object to model the behaviour of real world entities which change their status over time. This is a severe limitation in the context of a database programming language. Besides this, traditional object-oriented languages do not model the fact that the behaviour of real world entities may depend on the role that the entity plays in a context. We propose a mechanism to face both problems in the context of a statically strongly typed object-oriented database programming language, and show that the two problems are strictly related. We show that the problem can be solved without giving up the most useful features of object-oriented programming, namely inheritance, late binding and encapsulation. Examples will be given referring to the prototype implementation of the language.

1 Background

One of the major problems encountered in the maintenance of a database application is how to manage changes. We share completely the opinion of Richardson and Schwarz expressed in [7]: "Most object-oriented database systems display serious shortcoming in their ability to model both the dynamic nature and the many-faceted nature of common real-world entities. The most obvious example of this kind of entity is a person. While existing OODBSs may capture the notion that a student is a person, they do not support the notion that a given person may become a student. After graduation, that person ceases to be a student, and becomes an alumnus in the meantime, he or she may also be an employee, a customer, a club member, etc. Throughout his or her life, a person gains and loses many roles."

This problem has been investigated in the object-oriented database community by several authors, and we will comment on related works later on. The main contribution of this paper is the extension of an object-oriented data model with the notion of objects with roles, such that an object can have several roles and is always accessed through one of its roles. The behaviour of an object depends on the role used to access it. Moreover this mechanism is supported by a strongly typed programming language Fibonacci which also offers other features such as: a) the separation between the object interface, or type, and its implementation, to allow the evolution of the implementation without affecting the rest of the system which is only aware of the object interface; b) the possibility of having different implementations for a unique object type; c) the use of an inclusion hierarchy with multiple inheritance to organize object types. Besides objects, the data model provides also a class and association mechanism to model databases, but the presentation of these mechanisms is outside the scope of this paper and can be found in [2].

The paper is organized as follows. Section 2 describes the features of the proposed mechanism for objects with roles in a language independent fashion. Section 3 presents an overview of the Fibonacci type system to give the prerequisite to understand in Section 4 the constructs of the language to define objects with roles according to the requirements defined in Section 2. Section 5 compares the proposed solution with related works. In the conclusions, we comment on our future plan.

2 The features of the Fibonacci object mechanism

Real-world entities with roles. When constructing a computerized information system, adopting a simplified point of view, we will assume that the reality consists of entities, with certain *behaviours*, which evolve over time. Entities can play several *roles* during their life, i.e. they can belong to several conceptual categories. For example a human being may be classified as a person, an employee, a teacher, a department chairman, a tennis player, a retired employee, etc. In general an entity can have at same time several roles, although there are cases where some roles cannot co-exists (e.g., a person cannot be an employee and unemployed at the same time). However, any interaction with an entity always takes place through one specific role of the entity, and the behaviour of the entity may depend on

[1] This work has been supported in part by grants from the C.E.C. under ESPRIT BRA No.6309 (FIDE2: Fully Integrated Data Environment), and the Progetto finalizzato "Sistemi informatici e calcolo parallelo" of C.N.R. under grant No. 92.01561.PF69.

Proceedings of the 19th VLDB Conference
Dublin, Ireland, 1993, Rakesh Agrawal, Seán Baker, David Bell editors, Morgan Kaufmann Publishers, San Mateo, CA, USA. pp. 39-51.

the role it is playing. The set of roles possessed by an entity can change over time, so that its behaviour changes over time too.

Objects and messages. An object is the computer representation of a real-world entity. An object is a software entity which has an internal state equipped with a set of local operations (*methods*) to manipulate that state. The request to an object to execute an operation is called a *message* to which the object can reply. The state of an object can only be accessed and modified through operations associated with that object (*state encapsulation*). Each object can send messages to itself (*self-reference*) and so it is able to activate his own methods (*self-recursion*). The message interpretation (i.e. choosing the method to activate to reply a message) always depends on the object that receives the message.

Each object is distinct from all other objects and has an identity that persists over time, independently of changes to the value of its state. For instance, the object representing the person John is different from any other object representing another person, but will remain the same even if his address or some other attribute changes.

Roles. Each objects has a set of *roles*. An object is not manipulated directly, but always through one of its roles, so that we either say that a message is sent to an object through one of its roles or, more simply, that *messages are sent to roles*. The answer of an object to a message may depend on the role which receives it. For example, an object with role Graduate will answer to the message Introduce with "I am John Smith, graduate in Computer Science at the University of Toronto", but in another context it may be used with the role Manager and then the answer at the same message will be "I am John Smith, manager of the marketing division". In traditional object-oriented languages an object cannot show this kind of behaviour.

Since messages are sent to roles, the set of messages which an object can answer to, is not described by its object type but by the role types of the roles of the object. In this sense, role types are very similar to object types of other object-oriented languages.

Finally, roles (i.e. objects accessed through a role) are denotable and expressible values of the language (first-class values). They can be assigned to variables, used as data structure components and as parameters or results of functions.

Modeling roles and behaviour evolution. Many kinds of entities throughout their life change their role and behaviour. An unemployed can become an employee, and then a manager. Consequently, to model naturally entities that evolve dynamically, they must be represented by objects that can change their set of roles without affecting their identity. In traditional object-oriented languages an object cannot show this kind of behaviour because objects have an immutable type throughout their life.

Role type hierarchies. A subtyping relation is defined on role types (we say either that R_2 is a *supertype* of R_1 or that R_1 is a *subtype* of R_2). This relation is asymmetric, reflexive and transitive. A role subtype can have several role supertypes, from which inherits all properties (*multiple inheritance*), unless they are explicitly redefined in the subtype (*overriding*); besides, a role subtype can add new properties. Properties of the supertype can only be redefined in a controlled fashion so that a value of the subtype R_1 can be used in all contexts in which a value of the supertype R_2 is expected (*inclusion polymorphism*).

In figure 1 a role type hierarchy is represented, where Student, Graduate and Employee are all subtypes of Person. Associated to each type are some specific (i.e. not inherited) properties of the type. The Department property in Student and Employee has a different meaning: it is the student's major department and the department where the employee works.

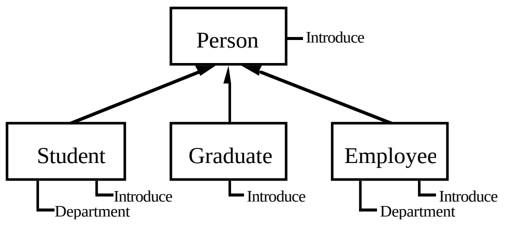

Figure 1. A hierarchy of role types

Separation between object interface and object implementations. A role type describes only the *interface* for objects with that type, i.e. the signatures of their methods. The implementation of the objects is given separately and objects with the same role type can have different implementations. This distinction between interfaces and implementations allows the creation of instances of the same type with different *structure* and *behaviour*. Other advantages of this approach will be discussed later on.

Normal interpretation of messages. Objects can acquire new roles during their lifetime and therefore new methods. Consequently, in general, an object X, in a certain point of its lifetime, may have several versions available for the method M to use in answering the message M (e.g. the Department as a Student and the Department as an Employee), and so it must decide which version of M it must use. In Fibonacci the decision is made by the role which receives the message. A detailed description of this decision process will be given in section 4; here we will only describe the basic ideas:

– *the behaviours depend on the role which receives the message*;
– *there are no interferences between "cousin" roles*[2]: the method to reply a message is chosen between the methods of the addressed role, including those inherited from its ancestors, or between the methods of the present subroles: no overriding is possible between cousin roles;
– *the most specialized behaviours prevail*[3]: for example, between Introduce of Person and Introduce of Employee the last property is prevailing; this corresponds to the classical *late binding* mechanism;
– *the behaviours become more specialized when time goes on*: when among the methods to choose there is not a

2 With respect to a fixed role, all the other roles in an object are either *ancestors* or *descendants* or *cousins*.

3 A metod M defined in the type T1 is a specialization of the method M in the type T2 if T1 is a subtype of T2.

most specialized one, the most recently acquired one is chosen; e.g. let us suppose that the Introduce message is sent to a person which is both Student and Graduate, then between Introduce of Student and Introduce of Graduate the last method is chosen.

Strict interpretation of messages. Fibonacci provides an alternative binding mechanism (*strict binding*), to observe the behaviour of an object in a certain role without taking into account possible specializations of that role. For example, we may send the message Introduce to a Person which is a Student too. With the normal binding mechanism, this causes the activation of the method Introduce of Student; whereas, with the strict binding mechanism, the invoked method will be that of Person. Strict binding allows to simulate the classical *send-to-super* mechanism of object-oriented languages, but it is a more general and flexible tool because it allows the activation of any method of any role.[4]

Roles visibility. As we can ask to an individual if he is a medical doctor and if so to behave as such, so in Fibonacci it is possible to query an object to know its current roles (*role inspection*) and to change the role through the object is accessed (*role casting*).

It is important to say that, despite of the richness of the model, if object extension and role casting are never used, the Fibonacci objects behave exactly the same as classical Smalltalk objects, and strict interpretation of messages coincides with normal interpretation.

Objects can be created but not destroyed. Objects are not destroyed explicitly, but they are eliminated automatically when they become *garbage*, that is they are no longer reachable from any variable or parent object in the programming environment.

A graphical representation of objects. Objects are seen by their users as a black box accessible by a set of roles (see fig. 2). Messages are sent to a role. Internally an object is made of two components (see fig. 3): a) a set of blocks where data and methods are stored, and b) a dispatcher in charge of directing the messages to the appropriate method that produces the answer (the dispatcher implements the dynamic binding of methods to messages). The blocks structure is not accessible to the users. The object identity is independent of its roles, of the content of the internal blocks (data and methods) and of the dispatcher structure. Finally, figure 3 shows that an object can send messages to itself as to any other object.

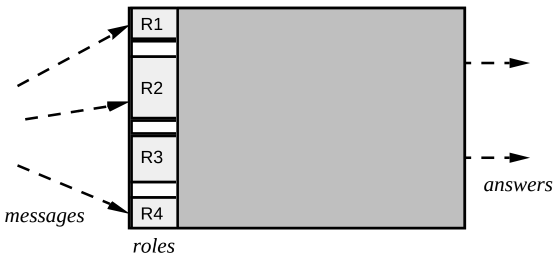

Figure 2. External view of an object

4 About *strict* vs. *static binding* see further sec. 4.7.

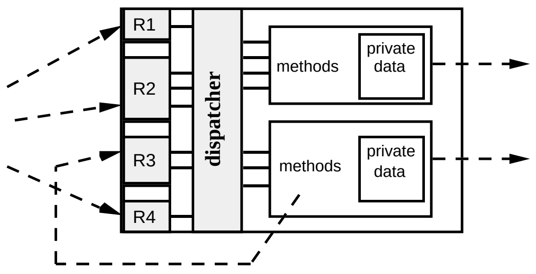

Figure 3. Internal view of an object

Figure 4 shows the internal structure of a simple object with the only role Person. The dispatcher structure changes when a new role is acquired by the object.

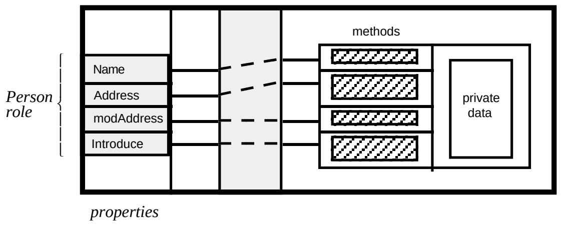

Figure 4. An object with the role Person

For example figure 5 shows how the object in figure 4 changes when it acquires the role Employee. In particular a new binding is created for the message Introduce of the role Person. The choice between the old and the new link depends on which kind of binding is required: the old link (the one with thinner dashes) is for strict binding, whereas the new one is for normal binding (for all the other messages, normal and strict binding coincide).

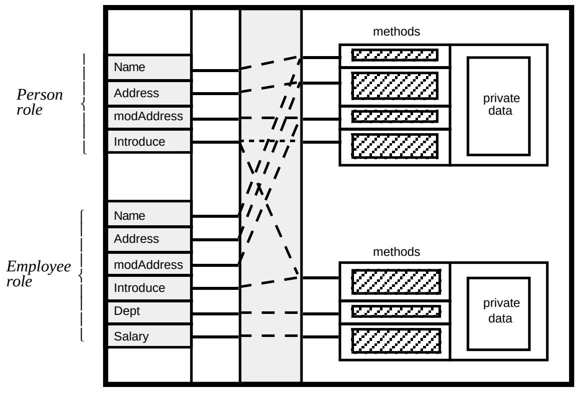

Figure 5. An object with the roles Person and Employee

3 An overview of Fibonacci

Fibonacci is an object-oriented database programming language descendent of the language Galileo [1].

Fibonacci is an expression-based, statically-scoped, functional (functions are first-class values), interactive and persistent language. The last property means that all data transitively accessible from the global environment (top-level), survive automatically between different worksessions, independently of their type. Data are removed by a garbage collector when they are no longer reachable from any identifier in the global environment.

Fibonacci is a strongly-typed language. Each legal expression has (at least) a type which is statically checked. Each type is related to a set of operators which can be applied to values of such type (e.g. the field selectors of a tuple type). The basic types are `Bool`, `String`, `Int`, `Real`, `Any`, `None`, and `Null`. Each basic type is different from other basic types and from all the user defined types. The instances of basic types are all disjoint, with a notable exception: the value `unknown` (of type `None`), which belongs to any type whatsoever.

A set of type constructors is provided to define new types: tuples, labelled variants, sequences, functions. These type constructors take types as parameters, and produce other types on which the equality is structural (i.e. two types are equal if they are built with the same constructor applied to types recursively equal). Basic and constructed types will be referred, in the sequel, as *concrete types* to distinguish them from *object* and *role types*.

The type constructor `Var` applied to a type T return the type of variables of type T. On such type are defined the usual assignment operator (:=), and an explicit dereferencing operator (`at`). The value constructor `var` applied to a value v of type T, return a variable cell containing the v value.

Values of concrete types share the following important properties:

– the equality on them is structural (two values are equal if they are of the same type and have recursively equal components), except for functions and modifiable values, on which equality is defined as identity (sameness);
– they are used *directly*, and not by copying them, when they are passed as parameters to functions, bound to identifiers in declarations, and used in constructing complex values.

An *implicit* subtype relation is defined on concrete types. This relation allows the so called *inclusion polymorphism* to be exploited: if T1 is a *subtype* of T2 (also, T2 is a *supertype* of T1), then a value of T1 is also a value of T2, consequently, it can be used in every context where a value of T2 is expected. A subtype relation holds also among role types when it is explicitly declared. `None` and `Any` are respectively the bottom and the top of the type hierarchy.

4 The Fibonacci object mechanism

In this section we will present the constructs which realize the model informally described in section 2.

4.1 Object and role types definition

The most peculiar feature of Fibonacci's object model is the distinction between object and role values. In Fibonacci objects are not directly manipulated, but are always accessed through one of their roles. Hence, role values and role types are used in Fibonacci to accomplish all the operations usually related to objects and object types. For this reason, we will often say "the object r" instead of "the role r of the object o".

At the value level, roles answer to messages while objects, essentially, retain the identity of a set of roles. Referring to fig. 2, the object is the box while the roles are the entry points for the object. In fact the only operators available on objects are equality, extension with new roles, role inspection and role casting (see sec. 4.7 and 4.8). For these operations, the involved object is denoted by specifying one of its roles (the specific role chosen is irrelevant).

`NewObject` is the constructor for a new object type which is the supertype of all its role types, i.e. its *role type family*. Since messages are always sent to roles, and not directly to objects, the set of messages which an object can answer to is not specified in the object type but in the role types of the corresponding role type family.

For example, a definition of a new object type PersonObject is:

```
Let PersonObject = NewObject;
```

`NewObject` is a generative type definition: every time it is used a new object type, different from any other, is defined.

A role type is defined with the constructor **`IsA`** … **`With`** … **`End`** as a subtype of an object type or as a subtype of other role types. A role type is defined by a set of *properties* which defines the method signature for its values. **`IsA`** … **`With`** … **`End`** is a *generative* operator; it produces a new type, different from any other type, each time is used.

```
Let Person = IsA PersonObject With
       Name: String;
       BirthYear: Int;
       Age: Int;
       Address: String;
       modAddress (newAddress: String): Null;
       Introduce: String;
       End;
>>> Let Person <: PersonObject = <Role>
```

Figure 6. The Person role type

Figure 6 shows the definition of the role type Person, entered interactively at the top-level, and the system answer.

The **`Let`** keyword precedes a type declaration. Fibonacci adopts the lexical convention by which all type constructs and predefined type names are capitalized. **`IsA`** `<Type>` **`With`** `<properties list>` **`End`** is the type constructor for role types. The semicolon terminates a phrase (declaration or expression). The symbol >>> precedes the system answer to the type definition. The symbol <: denotes the subtype relation.

4.2 Object construction

A role type T defines the interface of the objects with such a type, but doesn't give information about their internal structure. An object with a role type T is created with the construct **`role`** `T <implementation>` **`end`**, where the implementation specifies the private state of the object and the body for all the methods specified in the interface.

Figure 7 shows the implementation for an object named john with a role type Person.

```
let john = role Person
    private
        let Name = “John Daniels”;
        let BirthYear = 1967;
        let Address = var
                    (“123, Darwin road – London”);
    methods
        Name = Name;
        BirthYear = BirthYear;
        Age = currentYear() – BirthYear;
        Address = at (Address);
        modAddress (newAddress: String) =
                            Address := newAddress;
        Introduce = “My name is ” & Name &
                    “ and I was born in ” &
                    intToString(BirthYear);
    end;
>>> let john : Person = <role>
```

Figure 7. Single object construction

The **`let`** keyword precedes a value declaration, which bounds a role of a newly created object to john. The evaluation of the expression **`role`** `T` **`private`** `<private env>` **`methods`** `<methods env>` **`end`** creates a new object and returns a role of type T for that object; we say, more simply, that it creates a role. `<private env>` is a *sequence* of declarations or expressions, evaluated once when the object is constructed. Each declaration or expression has visibility of the preceding ones. `<methods env>` is a *set* of method specifications, i.e. all method names are different and their order is not significant. A method is specified by giving its name, its signature (compatible with the expected signature) and its body (the expression following the = symbol). All methods declared in the interface must be specified.

The evaluation of the expression **`role`** `T` **`private`** `<private env>` **`methods`** `<methods env>` **`end`** involves the following steps:

– the declarations in `<private env>` are *sequentially* evaluated to create a private environment on top of the current external environment (in the example, the top-level environment);
– this private environment is extended with the *code* of methods defined in `<methods env>`; even parameterless methods are not evaluated at object construction but only when they are called; methods may refer private or external data and also the whole object being built, through the `me` identifier (see further);
– a new object is created which contains the methods, the private environment and the interface for role T (now we can say that the object *has* the role T);
– the interface is connected to the appropriate methods;
– the specified role of the newly created object is returned.

Figure 8 shows the structure of the object john resulting from the evaluation of the declaration shown in figure 7.

Once an object is created, its methods can be selected with the dot notation. Method call causes the evaluation of an expression in the private environment with possible side-effects; this is the only way to ask an object to modify its internal state. Examples of method call are:

```
john.Address;
>>> “123, Darwin road – London” : String

john.modAddress(“Beagle – Pacific Ocean”);
>>> nil : Null

john.Address;
>>> “Beagle – Pacific Ocean” : String
```

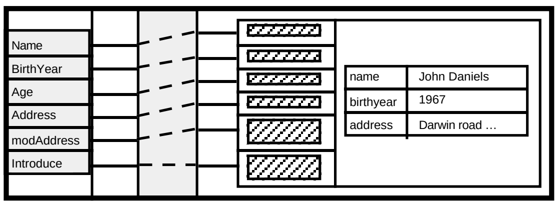

Figure 8. Inside the john object

4.3 “`const`” and “`mod`” properties

In object-oriented database applications, most messages are used only to retrieve and update the value of a variable hidden in the state of the object. It is important to give a special status to these messages for three reasons:

– *documentation*: giving a declarative way to specify in the interface of an object that some methods are meant to be used as specified above improves program readability, like any information about the expected behaviour of a method does;
– *usability*: giving an easier way to implement this common class of methods helps the programmer;
– *implementation*: if the system knows that, for all objects in a type, a given message just accesses a variable in the state, it may exploit this kind of information to build an index over that component of object state, to improve the response time of queries involving that message.

Many languages face this issue by giving direct visibility to object state, or to a part of it. In Fibonacci, when messages are meant to be used just to access an object state, this information can be specified, without breaking encapsulation, as follows.

A property of an object role type can be defined **`const`** to mean that the value returned by the corresponding method is always the same, as long as the object is not extended into a subtype; a **`const`** property does not have parameters. Moreover when two properties `M:Type` and `modM(newM: Type): Null` are related by the fact that the second property is used to modify the value returned by the first one, the abbreviation **`mod`** `M:Type` can be used to define both of them. This declaration also asserts that the value returned by the `M` method is always the same until a `modM` method is called. According to these abbreviations the definition in figure 9 is equivalent to that in figure 6.

```
Let Person = IsA PersonObject With
        const Name: String;
        const BirthYear: Int;
        Age: Int;
        mod Address: String;
        Introduce: String;
        End;
```

Figure 9. Another Person role type

More precisely, both definitions produce the same method signature, but only the second one imposes some constraints on the behaviour of methods Name, BirthYear and Address.

The implementation of a role type with **`const`** and **`mod`** properties is simplified since the system provides a *standard implementation* for these properties. For a property **`const`** `P:TP` it is sufficient to declare in the private environment a value P of type TP' <: TP. Then, if a method named P is not declared, the standard implementation is automatically defined as: `P = P`. For a property **`mod`** `V:TV`, a private variable V of type TV must be declared in the private environment; then, the standard methods are: `V = at V` and `modV(newV:TV) = V:=newV`. With the standard implementation, the example 7 can be rewritten as shown in fig. 10.

```
let john = role Person
    private
        let Name = “John Daniels”;
        let BirthYear = 1967;
        let Address = var (“123, Darwin road –
London”);
    methods
        Age = currentYear() – BirthYear;
        Introduce = “My name is ” & Name &
                    “ and I was born in ” &
                    intToString(BirthYear);
    end;
```

Figure 10. Another constructor for john

The standard implementation is just a facility for the programmer, which can always provide its own implementation for the messages, typically to check some constraints or to perform additional side-effects. But also when the implementation is explicitly defined, the system enforces the constraints implied by the **`const`** and **`mod`** declarations.

4.4 Definition of an object constructor

In the previous examples single objects have been built from scratch, but usually we are interested in creating, for each role type, many instances with the same internal structure and method bodies. The problem is solved by defining a *constructor*, that is a function which returns new objects with a certain role. An example is shown in figure 11.[5]

The expression **`fun`** `(<arguments>):<type>` **`is`** `<exp>` defines a function, with type **`Fun`** `(<arguments>):<type>` and body `<exp>`.[6] When the function is applied, a new instance of Person is created. While the private data are different for each instance, the method bodies are shared by all instances.

In the body of the Introduce method the special identifier `me` denotes the constructed object. The *formal type* [7] of `me` is the type of the **`role`** expression where `me` is used (in this example Person); `me` can be used only in the method bodies and in the **`init`** expression. `intToString` is a predefined function to convert an integer into a string. The infix operator & is the concatenation operator on strings.

```
let createPerson = fun (name, address: String;
birthyear: Int) : Person is
    role Person
    private
        let Name = name;
        let BirthYear = birthyear;
        if stringLength(address) < 2 then
                failwith “incorrect address” end;
        let Address = var (address);
    methods
        Age = currentYear() – me.BirthYear;
        modAddress (newAddress: String) =
            if stringLength(newAddress) < 2
            then failwith “incorrect address”
            else Address := newAddress end;
        Introduce = “Name: ” & Name & “ – Age: ”
                    & intToString(me.Age);
    init
        if me.Age < 0 or me.Age > 150
        then failwith “incorrect birth year” end
    end;
```

Figure 11. A Person constructor

The clause **`init`** `<exp>` defines an expression which is evaluated when the object is built before returning it. In the expression the identifier `me` can be used as in a method body; as a matter of fact, the clause **`init`** may be seen as a special method evaluated once before returning the object. If the expression fails, the object construction fails and the effects are undone, since object creation is atomic.

Lets us see some examples.

```
let paul = createPerson(“Horace De Saussure”;
“Geneva”; 1960);
>>> let paul : Person = <role>

paul.Introduce;
>>> “Name: Horace De Saussure – Age: 33”: String

paul.modAddress(“”);
>>> failure: “incorrect address”

let dante = createPerson(“Dante Alighieri”;
“Ravenna”; 1265);
>>> failure: “incorrect birth year”
```

4.5 Role type hierarchies and inheritance

An object role family can be extended dynamically by defining a new role type T as a *subtype* of others, called its *supertypes*. The subtype *inherits* all properties of its supertypes, unless they are explicitly redefined in the subtype (*overriding*). In case of multiple inheritance, if a property is present in more supertypes, and there is not an explicit redefinition in the subtype, then the property of the *last specified supertype* is inherited, but only if that property has been defined in a common ancestor.

Figure 12 shows the definition of Student and Employee, both subtypes of Person.

In a subtype definition S, for any property P of S

---

5 This approach to the specification of object constructors is similar to the one adopted in Emerald [6].

6 A function definition has a syntax different from that of a method to reflect the fact there are differences between functions and methods: a function is a first class value, and so can be passed as parameter or returned as value by a function; a method is not a value, and it can only be evaluated by the object to which belongs for side effects or to return a value.

7 Because of subtyping, the type of an expression is generally just a supertype of the type of the values which will correspond to that expression at run time. For example, if the *x* parameter of a function has type *Person*, then it may be bound, at run time, to values belonging to any subtype of *Person*; in this case we say that *Person* is the *formal* type of *x*.

(inherited, redefined or added), if P is also defined in the supertype T then the following conditions hold:

– the signature of P in T is a subsignature of that in S (*contravariance*); [8]
– the output type of P in S is subtype of that in T (*covariance*);
– if P is neither **const** nor **mod** in T, then P in S may be declared as **const** or **mod**;
– if P is declared as **const** in T then the same must be in S;
– if P in T is declared as **mod** `P:TP`, then P must be declared as **mod** `P:TP` also in S.

The rule that a mod property cannot be redefined by specializing its type is a consequence of the fact that the signature of a redefined method must be contravariant. In fact the declaration **mod** `P:TP` introduces a method `modP(TP): Null` and the redefinition **mod** `P:TP'`, with TP' subtype of TP, introduces a method `modP(TP'):Null` which violates the contravariance rule for functional components.

```
Let Student = IsA Person With
       mod Faculty: String;
       const StudentNumber: Int;
       Introduce: String;
       End;
Let Employee = IsA Person With
       mod Department: String;
       const EmployeeNumber: Int;
       Introduce: String;
       End;
```

Figure 12. Student and Employee role types

4.6 Subtype object construction

When a role type is defined by inheritance, a constructor for objects belonging to that role may be either defined from scratch or by inheritance, i.e. by extending a constructor defined for a supertype. In this section we exemplify the first approach, while the second, which is more standard, is described in section 4.8. Figure 13 shows the direct (no inheritance) definition of the Student constructor.

To construct an object with role type T from scratch, the method for each property must be specified. The constructed object will have the role type T and all the supertypes of T. For example, with the following declaration:

```
let spinoza = createStudent(“Bento d’Espinoza”;
“Cordoba”; “Philosophy”; 1966);
```

is created the object `spinoza`, shown in figure 14.

[8] A signature is a list of zero or more pairs `Identifier: Type` separated by semicolons. We say that S1 is a *subsignature* of S2 if S1 extends S2 with new pairs or redefines (in the same order) the S2 pairs with more specialized types.

```
let createStudent = fun (name, address,
faculty: String; birthyear: Int) : Student is
   role Student
   private
      let Name = name;
      let BirthYear = birthyear;
      let Address = var (address);
      let Faculty = var faculty;
      let StudentNumber = newStudentNumber();
   methods
      Age = currentYear() – me.BirthYear;
      Introduce = “Name: ” & Name & “ – Age: ”
            & intToString(me.Age) &
            “ – Faculty: ” & me.Faculty;
   init
      if me.Age < 18 or me.Age > 70
      then failwith “incorrect birth year” end
   end;
```

Figure 13. A Student constructor

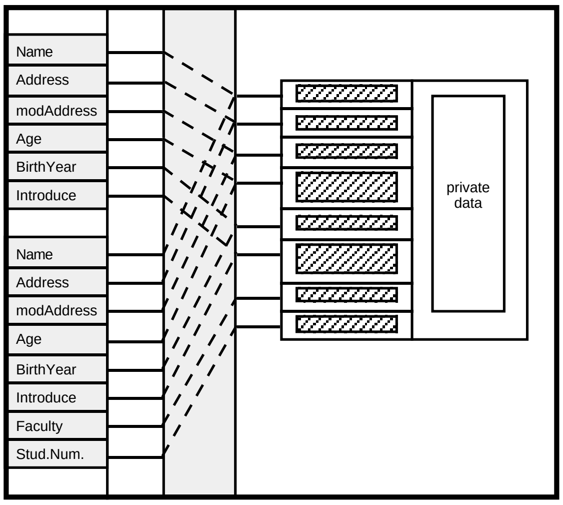

Figure 14. An object with two roles which share the same implementation

4.7 Other operators: object comparison, role inspection, role casting and strict binding

The language provides the following operators on objects:

– the equality operator (=) to test if two objects are the same, independently of their current role type; for example

```
john = spinoza;
>>> false : Bool
```

– the infix predicate **isAlso** to test if an object has a certain role; for example:

```
spinoza isAlso Person;
>>> true: Bool

john isAlso Student;
>>> false: Bool
```

– the infix operator **as** to coerce an object to one of its possible roles (*role casting*). The operator will fail if the object does not have the specified role:

```
let baruch = spinoza as Person;
>>> let baruch : Person = <role>

baruch = spinoza;
>>> true : Bool

let johnAsStudent = john as Student;
>>> failure: "as"
```

The expression `x as T` is well typed if T and the type of x belong to the same role type family.

The following operators are on role values:

– the infix predicate **`isExactly`** to test the actual type of a role value:

```
spinoza isExactly Student;
>>> true : Bool
```

– the infix operator '!' to request an object role to evaluate a method without considering the possible redefinitions of the method in its subroles (*strict binding*). This operator is useful, for example, to see the behaviour of a Person independently of the fact that he may also be an Employee or a Student (examples will be given in sec. 4.8).

Strict binding should not be confused with *static binding*: static binding takes place at compilation time and the method to activate is chosen on the base of the formal type of the expression which denotes the receiver of the message. Strict binding, which is a kind of dynamic binding, takes place at run-time and the method to activate is chosen depending on the *actual type* of the receiver. The type checker will guarantee that the actual type is a subtype of the formal type.

The combination of strict binding with role casting (e.g. `(X as T)!P`) is a useful feature of Fibonacci, in that: a) it allows to simulate static binding, b) it allows to simulate the traditional *send-to-super* mechanism of object-oriented languages (see sec. 4.8), c) in extension operators, it allows the programmer to specify explictly from which ancestor a method implementation is inherited.

4.8 Dynamic object extension

To model the *role and behaviour evolution* of entities, Fibonacci provides an *extension operator*, which allows an object to be extended dynamically with new subroles. Figure 15 shows the extension of john from Person to Student.

```
let johnAsStudent = ext john to Student
   private
       let Faculty = var "Science";
       let StudentNumber = newStudentNumber();
   methods
       Introduce = (me as Person) ! Introduce &
             ". I am a Science student";
   end;
>>> let johnAsStudent : Student = <role>

john = johnAsStudent;
>>> true : Bool
```

Figure 15. john becomes student

The object john acquires the role Student without changing its identity (as results from the test `john = johnAsStudent`). Note the combination of role casting with strict interpretation to call the method Introduce defined in Person. The object johnAsStudent is represented in figure 16 (compare it with the representation of john given in figure 8). Note the twofold link for Introduce: the old link is chosen for strict binding, whereas the new one for normal binding. For example, let us see how the behaviour of john has changed after the extension:

```
john.Introduce;
>>> "My name is John Daniels and I was born in
1967. I am a Science student" : String

johnAsStudent.Introduce;
>>> "My name is John Daniels and I was born in
1967. I am a Science student" : String

(johnAsStudent as Person)!Introduce;
>>> "My name is John Daniels and I was born in
1967" : String
```

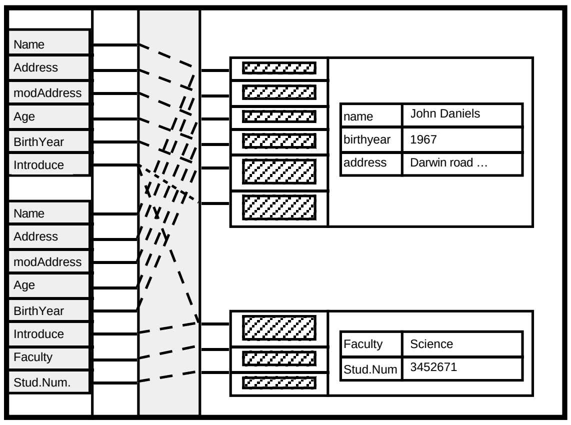

Figure 16. The internal structure of john after the extension

To explain the difference between the creation of an object from the scratch and by extension, it is useful to compare graphical representation in figure 16 with that in figure 14.

The construct **`ext`** has an header (**`ext`** `<object>` **`to`** `<target types>`) and an implementation (**`private`** … **`methods`** … **`init`** …). The implementation part is identical to that of the role operator (see sec. 4.2), while the following differences appear in the header part:

– `<object>` is an expression which denotes the object to be extended.
– `<target types>` are the role types that must be acquired by the object. The order in which are listed determines the order in which the roles are acquired. The last specified (called *target-type* of the extension) must be a subtype of all the previous ones. All the target types must belong to the same role family to which the type of `<object>` also belongs.
– the methods defined in the **`methods`** section must be at least those explicitly specified in the interfaces of the target types.

Let R1 and R2 be role types such that R1 <: R2, the object `X` is called *complete* if `X` **`isAlso`** `R1` implies `X` **`isAlso`** `R2`. Static and dynamic tests ensure that the extension operation always produces *complete objects without duplicate roles*. Figure 17 shows the definition of an extension operator to obtain an Employee from a Person.

The figure 18 shows how the behaviour of john changes once it has acquired the role type Employee:

```
let toEmployee = fun (aPerson: Person; dept:
String) : Employee is
    ext aPerson to Employee
    private
        let Department = var (dept);
        let EmployeeNumber =newEmployeeNumber();
    methods
        Introduce = (me as Person) ! Introduce &
        ". I am an employee";
    end;
```

Figure 17. An extension operator

```
toEmployee(john; "Quality Management");

john.Introduce;
>>> "My name is John Daniels and I was born in
1967. I am an employee" : String
```

Figure 18. john becomes Employee

The behaviour of john as Student does not change once it acquires the type Employee:

```
johnAsStudent.Introduce;
>>> "My name is John Daniels and I was born in
1967. I am a Science student" : String
```

Implementing constructors by inheritance

Using the constructor createPerson and the operator toEmployee it is possible to define a constructor createEmployee which makes use only of predefined implementations:

```
let createEmployee =
    fun (name, address, dept: String;
            birthyear: Int) : Employee is
    toEmployee(createPerson(name; address;
            birthyear); dept);
```

Another way to reuse the implementation of createPerson is shown in figure 19.

```
let createStudent2 = fun (name, address,
faculty: String; birthyear: Int) : Student is
    ext createPerson(name; address; birthyear)
    to Student
    private
        let Faculty = var (faculty);
        let StudentNumber = newStudentNumber();
    methods
        Introduce = (me as Person) ! Introduce &
        ". I am a student of " & me.Faculty;
    end
```

Figure 19. Reusing a Person constructor to create students

Note that a role type can have multiple constructors, and that in defining a constructor for a role subtype it is possible to choose which super-role constructor is extended.

Object extension and multiple inheritance

Let us define the type TeachingFellow to show other examples of multiple inheritance and object extension.

```
Let TeachingFellow =
        IsA Student, Employee With
            const Course: String;
            Introduce: String;
        End;
```

Figure 20 shows an operator to make a TeachingFellow from a Student:

```
let fromStudentToTeachingFellow =
    fun (aStudent: Student; dept,
            course: String) : TeachingFellow is
    ext aStudent to Employee, TeachingFellow
    private
        let Department = var (dept);
        let EmployeeNumber =NewEmployeeNumber();
        let Course = course;
    methods
        Introduce = (me as Student)!Introduce &
                   "– Course: " & course;
    end;
```

Figure 20. An operator to make a TeachingFellow from a Student

The interesting aspect in the example is that there are two roles to be acquired: the first (Employee) is not a subtype of Student, while the second role is a subtype of Student, and so the condition is satisfied that the target-type must be subtype of those which precede it. Let us show how the extension operation changes the behaviour of the object to be extended:

```
fromStudentToTeachingFellow(spinoza; "Hermetic
Philosophy"; "Ethica");

spinoza.Introduce;
>>> "Name: Bento d'Espinoza – Age: 27 – Faculty:
Philosophy – Course: Ethica" : String

(spinoza as Employee).Introduce;
>>> "Name: Bento d'Espinoza – Age: 27 – Faculty:
Philosophy – Course: Ethica" : String
```

4.9 Object contraction (role dropping)

In order to meet the need for modelling roles and behaviours evolution, Fibonacci should also provide a *contraction operator*, i.e. a mechanism to allow the objects to lose some roles (e.g. **`drop`** `R1,R2` **`from`** `X`). With such an operator one could model, for instance, the fact that when a student takes a degree loses his Student role and gains the role of Graduate, or the fact that a worker at the end of his career loses the Employed role and becomes a Retired.

The Fibonacci's contraction mechanism should have the following features:

– when a role is dropped from an object, all its subroles are lost too;
– the objects are not destroyed (there are no dangling references);
– casting toward a dropped role (e.g. `X` **`as`** `R`) arises a

trappable failure, thus no one can take new acquaintance of a dropped role (no new reference to it can be created after the dropping);
– sending a message to a dropped role arises a trappable failure (*message passing failure*);
– role inspection and equality still work on a dropped role, since these operators refer to the object, rather than the roles;
– when a role is dropped from an object, previously hidden behaviours are brought in the foreground; e.g. if john loses the Employee role, his answer to `(john **as** Person).Introduce` will be again that of Student;
– role dropping is an important event in the life of an object; then such a state transition should be monitorable through preconditions expressible in the implementation (like the **`init`** clause).

It is important to notice that the *message passing failure* in Fibonacci is different from that of other object-oriented languages (firstly Smalltalk): in Fibonacci the failure informs the sender that the receiver has lost a role; whereas, in languages with dynamic type checking, this failure only represents a wrong use of an object.

Role dropping is an operation similar to object removal, thus the well known problem of the *referential integrity* should be taken in account [7].

To model the fact that not every sequence of role acquisitions or role losses is admissible, it should be possible to specify *admissible histories* or *migration paths* in a role type hierarchy (sequences of **`ext`**/**`drop`**) [10].

These problems are not dealt with in the currrent implementation of Fibonacci, but we are working on them to provide the language with a contraction operator.

4.10 Message interpretation

The role mechanism is essential when objects can be extended with independent subroles. In this case, classical late binding without roles creates a problem. Suppose that a type Person has two different subtypes Student and Employee, and that both of them add a property PersonalCode to the supertype. The two personal codes have unrelated semantics, and maybe even a different type. Let john be created as a Person and later on extended, first to Student with code 100200 and then to Employee with code "jhn698". In a language with late binding and without roles, johnAsStudent answers "jhn698" to a message PersonalCode, or johnAsEmployee answers 100200, because the objects always exhibit a uniform behaviour. This is both a semantic error and a type-level error. Since it is not known statically whether an object of type Student has also been extended to Employee, we can conclude that the system can never be sure that any object of type Student answers the message PersonalCode with an integer. More generally, if it is always possible to add new object types to the system, the type checker can never be sure of the type of the result of any message passing operation.

This problem may be faced by imposing constraints on methods appearing with the same name in cousin object types. This contrasts with the typical usage of object-oriented languages. In these languages, if some programmers work at the same time at the same project, any programmer is free to take general-purpose object types from libraries and specialize them, regardless of the fact that other programmers are producing cousin object types by specializing the same library for different purposes. Forbidding name duplications in all the possible specializations of a library object type would damage one essential abstraction mechanism of object-oriented programming. It could be likened to forbidding the usage of the same name for a local variable in two different unrelated functions. Preventing undesired interactions between cousin roles, to attain full “cousin role independence”, is one of the primary design choices of the message interpretation rules.

Message interpretation can be described as follows. When a role receives a message it first checks whether any of its descendants has its own method (not inherited) to reply to the massage. If such descendant is found, then it is *delegated* to answer the message. The descendants are tried in *reversal temporal order*, i.e. the last acquired descendant is tried first. Subtyping ensures that the delegated role can safely substitute the receiving one. If no delegate is found, the receiver searches an implementation for the message inside itself. If this is not found, then the receiver looks for an implementation for the message in the ancestor role from which the corresponding property is *inherited*. The typing rules ensure that this last search is always successful. Note that this is just a way to describe the meaning of message passing; alternatively, the same semantics can be described by specifying, with reference to Figures 3, 4 and 5, how the dispatching structure of an object is set up and how it is modified when an object is extended.

For example, the message Introduce sent to john (see figure 18) causes the activation of the Introduce method of Employee, because Employee is the last acquired subrole of the object, hence the method will be executed by *delegation*. The message Introduce sent to `johnAsStudent` will be answered by the method of Student, because there is no descendant of Student in `john`. Instead, if `johnAsStudent` receives the message Name the answering method will be that of Person, hence it will be executed by *inheritance*.

Self-reference semantic

The distinction between delegation and inheritance is essential to understand the meaning of self-references in the method body. The following rules apply: a) when a method M, belonging to the role R, is activated by delegation (in other words the receiving role is a superrole of R), the actual type of `me` in that activation of M will be just R (i.e. its formal type); b) when the same method is executed by inheritance (the receiving role is subrole of R) the actual type of `me` will be that of the role which originally received the message M.

Rule a) is essential to the *type safety* of the language. Let, indeed, RR be the receiving role of the message, let DR be the role delegated to answer (then DR <: RR); the formal type of `me` in DR’s method is DR, then to ensure a type-safe execution the actual type of `me` must be DR or a subtype of DR.

Rule b) is the classical rule adopted by object-oriented languages. Suppose for example that in a graphical editor an object type Picture is defined with a method Draw taking a color as a parameter. Squares and Circles are subtype of Picture, and contain the actual code for the Draw method. However, a method DrawBlack can be implemented once for all for the object type Picture, as `me.Draw(black)`. When a Square executes by inheritance the DrawBlack method, the Draw(black) message is sent to `me` seen as a Square.

It can be interesting to note that the rule b), besides being useful, is a consequence of the principle of *non-interference between cousins*. If `me` in a method which is activated by inheritance were bound to the role where the method is defined, then self-reference would allow methods of cousin roles to be activated. Let us consider the example in figure 21, where each method is associated with the corresponding body.

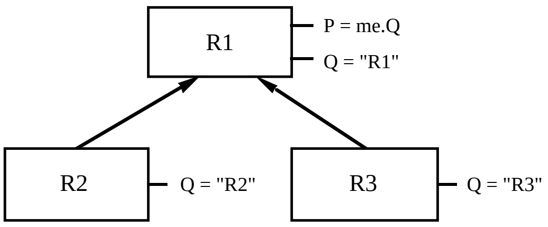

Figure 21. A role type hierarchy

Let us assume that the object X has been created with roles R1 and R2 and then extended with role R3. Adopting the correct rule to solve self-reference, when the message P is sent to X seen through the role R2, the answer is “R2”. If we had adopted the other rule (self-reference bound to the role which owns the method activated by inheritance), the answer would have been “R3”, and therefore the method of a receiving role (R2) would be *covered* by a method of a cousin role.

Final remarks

In traditional object-oriented languages all methods are executed either by the receiving role or by inheritance. This happens because the only role accessible of an object is the bottom role, which has no descendant. So we can affirm that both binding mechanisms of Fibonacci are a generalization of the classical late binding mechanism.

With respect to a fixed role, all the other roles in a Fibonacci object are either ancestors or descendants or cousins. The message interpretation mechanism ensures, in a word, that there is neither interference nor inheritance between cousins. This is very important, since in general when an object is extended with two cousin roles (e.g. a Person with Student and Employee), if the same method is defined in all the three roles, the two cousins can specialize it with two subtypes T’ and T’’ of the type T assigned by the father to that method, but there is no subtype relation between T’ and T’’, which implies that inheritance between cousins would be unsound not only with respect to the modelling principles, but also with respect to the language typing rules.

5 Previous works

In the last fifteen years the need for data modeling features capable of capturing the evolving and multifaceted nature of real world entities has been pointed out by many researchers. The first attempt in this direction was the *role model* of Bachman and Daya [4], aimed to enhance the expressive power of network data model. In more recent years, the Galileo language provides a mechanism to allow instances of a class to become, dynamically, instances of a subclass and, at the same time, to acquire new behavioral aspects without losing their identity [1]. This mechanism was found useful to model the behavioral specialization of world entities over their lifetime, but it has limitations because of the assumption that every object always belongs to a unique most specialized class (type). In what follows we review some of the more relevant recent proposals in the context of object-oriented database programming languages.

Iris

Iris [5] is an OODBMS equipped with explicit features to model behavioral evolution of entities. Iris objects may acquire or lose types during their life, retaining their identity; but is not possible to observe an object from different perspectives, indeed, despite type multiplicity, an object, in a fixed instant of its life, always exhibits a uniform behaviour, no matter the context from which is observed. For instance, suppose a property P is differently defined in types T1 and T2; then an object X, belonging to both of them, will always answer to the message P with the method of the most specialized type between T1 and T2. But if there is no such type the answer will depend on *ad hoc* rules which the user must establish to resolve such ambiguities. This approach is unsatisfactory because the type multiplicity cannot be used to model role multiplicity, and the objects show the behavioral uniformity typical of traditional object-oriented languages (i.e. Smalltalk). In addition, the resolution of ambiguities in message dispatching is left to the programmer, whereas, we believe it should be an important concern of the supported data model.

Clovers

Stein and Zdonik [9] propose a mechanism called *clovers* which allows to model entities with multiple and independent roles. The language which supports this mechanism has provision for strong type-checking and subtyping. With clovers an object created in a type T may become an instance of T’ subtype of T, acquiring methods and data specific of T’. The object behaviour depends strictly on the type through which the object is observed, and there is no *late binding*. Clovers provides also an operator for type inspection and two operators for type coercion: one to go up and one to go down in the type hierarchy, but without explicit mention of the target type. The main differences from Fibonacci are the lack of support for late binding, and the impossibility of explicitly referring the types to which one is interested.

Views

Shilling and Sweeney [8] present an extension of the object data model based on the concept of *view*. In that model, an object is equipped with multiple interfaces (views). Every interface has its own set of methods and the interfaces of an object are separated and independent each of the others; the object is always referred through one of them, so there is no conflict between methods with same name belonging to different views. Every interface has a distinct implementation and a distinct set of variables accessible only to its methods. The object behaviour depends on the interface used to access it, and the object identity is preserved across the various views; that allows one to model multiple and independent roles. That mechanism, on other hand, has no provision for late binding, inheritance and subtyping, moreover separation between interfaces and implementations is not supported.

Aspects

Richardson and Schwarz [7] propose a model whose objects may have multiple *aspects* (types) and may be extended with new ones during their lifetime, without losing their identity. Every aspect has its own methods and private data and an object is always referred through one of its aspects. The observed behaviour is that specific of the referred aspect and the late binding and inheritance mechanism are not supported. Interfaces are defined separately from implementations and the interface matching is structural, allowing to have more implementations for a given type, but also to reuse an implementation for more types. The type system has provision for an implicit subtyping relation (*conformance*). A new aspect added to an object X or to another aspect A of X, may hides some property defined for X; then there is no subtyping relation between an aspect and the type of the extended object. As already noted, the aspects proposal has no support for inheritance, neither single neither multiple. To overcome this limitation, an aspect B extending another aspect A, must explicitly replicate the A interface in its definition, and it must call the ancestor methods with a *send-to-super* primitive. Is not possible to extend an object with more aspects in a unique operation. Due to the structural matching between types, the aspects mechanism does not have operators for role inspection and role coercion.

Nuovo Galileo

In the data model proposed in [3] the objects can be dynamically extended with new types and are not constrained to have a unique minimal type, but the role mechanism is not provided. Then in order to support late-binding, the assumption is made that for each method a most specialized version of it always exists. Thus, the objects always exhibit a uniform behaviour, no matter the type through they are accessed. This object mechanism has been the first step in the development of the object mechanism of Fibonacci.

Summary

All proposals share the following features, found also in Fibonacci:
- objects may acquire new types and new behaviours;
- objects retain their identity during their life, no matter which extensions are operated and independently of the point of view through they are observed;
- encapsulation is preserved, because the extensions have no direct access to private data of the existing object.

A novel aspect of Fibonacci is, instead, the coexistence of late-binding and multiple inheritance with role multiplicity and dynamic object extension, in a framework with strong type-checking and subtyping. Moreover, the combination of such complex features is obtained neither to detriment of semantic clarity, neither relying on specification ambiguities which introduce implementation dependent or *ad hoc* semantics. Indeed, the full meaning of the various mechanisms is established at first in the data model and then substantiated in the constructs of the language.

Significantly, the proposals which support late-binding (Galileo, Iris and Nuovo Galileo), always assume the existence of a most specialized method in order to resolve the message dispatching ambiguities that can arise from type multiplicity. Vice versa, when the previous assumption is abandoned and objects are allowed to have multiple minimal types (Clovers, Views and Aspects), late-binding is never provided.

6 Conclusions

An object mechanism for a strongly typed database programming language has been presented. The object mechanism, besides the usual properties of state encapsulation, unchangeable identity, separate definition of interface and implementation, and late binding, has a role mechanism characterized by the following features:
- *plurality of behaviours*: a unique object can be accessed through different roles, which have different types and can answer in different ways to a message. Plurality of behaviours allows to model situations where a unique entity of the domain of discourse can play different roles and behaves in a different way according to its role. That relates roles to a view mechanism.
- *independence of extensions*: it is possible to perform two independent extension of a unique role with two cousin roles, without interference between them. Independence of extensions is especially helpful in the development of applications structured as independent modules.
- *strict and late binding*: the sender can choose between the two binding mechanisms, thus, it can decide whether delegation is allowed. The distinction between strict and late binding is most useful in implementing methods by extending or reusing existing implementations, and it is related to the *super* of most traditional object-oriented languages.
- *role casting and role inspection*: these are crucial features to fully exploit the richness of the object model; they allow one to navigate freely in the role graph of an object,

and to observe all its possible aspects and behaviours. These capabilities are very important in languages with strong typing, since they give, informally, the ability of dynamically changing the type of an object.

It is important to note that if a programmer does not use extension operators, but always builds objects in the subtypes using constructors, then there is no need to distinguish objects from roles, neither strict from late binding, and all the usual rules of object-oriented languages apply. So the complexity of the role mechanism comes into play only when really needed.

The object mechanism is one of the Fibonacci features designed to model object-oriented databases. The language provides also (a) a class mechanism to model a modifiable collection of values, on which it is possible to define an inclusion constraint, and (b) an association mechanism to model modifiable n-ary relations among classes [2]. All these features have been considered in the current implementation of a prototype of the language compiler.

References

[1] A. Albano, L. Cardelli and R. Orsini "Galileo: A Strongly Typed, Interactive Conceptual Language", *ACM Transactions on Database Systems*, Vol. 10, No. 2, pp. 230-260, 1985. Also in: *Readings in Object-Oriented Database Systems*, S.B. Zdonik and D. Maier (eds), Morgan Kauffman, San Mateo, California, pp.147-161, 1990.

[2] A. Albano, G. Ghelli and R. Orsini "A Relationship Mechanism for a Strongly Type Object-Oriented Database Programming Language", *Proc. of 17th Int. Conf. on VLDB*, Barcelona, 1991, pp. 565-575.

[3] A. Albano, G. Ghelli and R. Orsini "Objects for a Database Programming Language", *proc. of the third Intl. Workshop on Data Base Programming Languages*, P. Kannelakis, and J. W. Schmidt (eds), Morgan Kauffman, San Mateo, California, pp.236-256, 1992.

[4] C.W. Bachman and M. Daya "The role concept in data models", *Proceedings of the Third Int. Conf. on VLDB*, pp. 464-476, 1977

[5] D.H. Fishman et al. "Iris: An Object-Oriented Database Management System", *ACM Trans. on Office Information Systems*, vol. 5, n. 1, pp. 48-69, Jan. 1987.

[6] A. Black, N. Hutchinson, E. Jul, and H. Levy "Object Structure in the Emerald System", *OOPSLA '86, ACM SIGPLAN Notices*, pp. 76-86, Sept. 1986

[7] J. Richardson and P. Schwartz "Aspects: Extending objects to support multiple, indipendent roles", *Proceedings of the Int. Conf. on Management of Data, ACM SIGMOD Record*, vol. 20, pp. 298-307, May 1991

[8] J.J. Shilling and P.F. Sweeney "Three Steps to View: Extending the Object-Oriented Paradigm" *OOPSLA '89, ACM SIGPLAN Notices*, vol. 24, n. 10, pp. 353-361, Oct. 1989

[9] L.A. Stein and S.B. Zdonik "Clovers: The Dynamic Behavior of Type and Instances" Brown University Technical Report No. CS-89-42, Nov. 1989

[10] J. Su "Dynamic Constraints and Object Migration", *Proc. of 17th Int. Conf. on VLDB*, Barcelona, 1991, pp. 233-242.