# Design and implementation of an object-oriented, strongly typed language for distributed applications

*S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin, X. Rousset de Pina*

Bull-IMAG, 2 av. de Vignate, 38610 Gières, France

**1. INTRODUCTION** 1

**2. SYSTEM OVERVIEW** 2
- 2.1. Objects 2
- 2.2. Execution structures 3
- 2.3. Object location and storage 4

**3. OBJECT MODEL** 4
- 3.1. Types and classes 4
- 3.2. Type conformity 5
- 3.3. Subtyping and type restrictions 6
- 3.4. Subclassing and inheritance 8
- 3.5. Object composition 8

**4. SUPPORT FOR CONCURRENCY AND DISTRIBUTION** 9
- 4.1. Concurrent activities 9
- 4.2. Synchronization of shared objects 9
- 4.3. Distribution aspects 10

**5. IMPLEMENTATION** 11
- 5.1. Run-time representation of objects 11
- 5.2. Principle of the compiler 11

**6. EXPERIENCE** 12

**7. CONCLUSION** 14

# Design and implementation of an object-oriented, strongly typed language for distributed applications

*S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin, X. Rousset de Pina*

Bull-IMAG, 2 av. de Vignate, 38610 Gières, France

**Abstract**. This paper describes the design and implementation of a language for programming distributed applications. This language is closely integrated with the Guide distributed operating system, which provides its run-time support. The main features of the language are separate definition of types and classes, conform-ant subtypes, support for complex persistent objects, and location transparency. The language also provides concurrent activities and a synchronization mechanism for shared objects. The design of the language and the principles of its run-time support are presented. Early experience with the language is discussed.

## 1. INTRODUCTION

This paper describes the design and implementation of a language for the programming of distributed applications. This language is called Guide and its run-time support is provided by an object-oriented distributed operating system (also called Guide) designed and implemented at the University of Grenoble as a joint project of Laboratoire de Génie Informatique and Bull Research Center [Decouchant 88b]. The Guide system and language implement the distributed object-oriented architecture defined by Comandos (Construction and Management of Distributed Open Systems), a cooperative project under the ESPRIT European program [Horn 87, Alves Marques 88]. The objective of Comandos is to design and implement a platform for the programming of large and complex distributed applications which manipulate permanent data.

Recent work on distributed systems has emphasized the importance of a close integration between programming languages and operating systems. There are two main approaches to support distributed computation. In the first one, the operating system provides a set of primitives for interprocess communication, resource management, and access to remote information. These primitives are used by conventional programming languages, possibly through a specialized library. In the second approach, distributed computation is directly supported by a language. We have adopted this latter approach, which simplifies the task of the application programmer, since the main abstractions provided by the system are directly embodied in language constructs. The Guide programming language has been designed to support the object model on which the system is built; its constructs are carefully matched to those of the underlying system.

Several recent research projects have followed a similar approach. A recent survey of languages for distributed applications is given in [Bal 88a]. In the Eden system [Black 85a], applications were programmed in EPL [Black 85b], an extension of Concurrent Euclid; the design of Emerald [Black 86a, Black 86b, Jul 88] was based on the Eden experience. The Æolus language [Wilkes 86] was developed for the Clouds distributed system [LeBlanc 85]. The Orca language [Bal 88b] is being implemented to program applications in Amoeba [Tanenbaum 86, Mullender 87]. Finally, the Comandos project is currently developing its own programming language, called Oscar [Comandos 88]. Guide may be viewed as an exploratory implementation of some of the features being developed in Oscar.

The object model implemented in Guide is characterized by the following main features:

• Objects are typed. A type describes a behavior shared by all objects of that type, in terms of the signatures of the operations applicable to the objects. Types are statically checked; in some cases, a dynamic check is necessary.

• A class defines a specific implementation of a type. Classes are used to generate instances of objects. Several classes implementing a type may coexist; an implementation of a type may dynamically be replaced by another one.

• Subtyping defines a hierarchy of types. The subtyping hierarchy is paralleled by a subclassing hierarchy, with single inheritance. Generic types and classes are provided.

• Objects are persistent. They are named by system references (system-wide uniquely generated internal names). Complex structures may be constructed by embedding references to objects within other objects.

In addition, the language supports concurrency and allows to express synchronization constraints for the sharing of an object by concurrent activities.

The distinctive features of this work are a) the separation between type and class (which allows several distinct implementations of a type to coexist and to be dynamically replaced by each other), combined with a subtyping and subclassing scheme in which type-checking may be done statically in most cases; b) the support for concurrent activity execution and for the expression of synchronization constraints; and c) the mechanisms for the management of persistent objects and the construction of complex structures through mechanisms for object composition. In addition, distribution is supported by the underlying operating system and is not apparent at the language level (although it may be made visible if needed).

Object-oriented languages have attracted much attention in the recent years, especially in relation with databases and distributed operating systems. We believe that many concepts in these two areas could be unified through object models and languages, and our design is an effort in that direction. We briefly summarize some of the main related work.

• *Separation type/class and strong typing*. The introduction of strong typing in object-oriented languages is recent. Examples of typed object-oriented languages are Emerald, Trellis/Owl [Schaffert 86] and Eiffel [Meyer 88b]. All of them combine static type checking and dynamic binding. Explicit separate definition of types and implementations is present in Emerald and in the conceptual language Galileo [Albano 85] (in the form of "abstract" and "concrete" types).

• *Persistent objects*. Persistence is usually provided in languages for databases (FAD, Galileo) and distributed systems (Emerald). Other languages (Trellis/Owl, Eiffel) provide persistent objects as an add-on feature. In Guide, objects are persistent, but temporary objects of any type may also be defined.

• *Synchronization*. In Guide, synchronization constraints on shared objects are specified by control clauses which are part of the class definition. The expressive power of this mechanism is similar to that of synchronization counters [Robert 77] or path expressions [Campbell 74]. A transaction mechanism (not described) is also provided.

• *Object composition*. Our approach here is related to that of recent database programming languages. For instance, FAD [Bancilhon 87] relies on two constructors: tuples (records) and sets, and supports object identity. Our composition scheme through references provides a similar power.

The remainder of this paper is organized as follows. Section 2 gives an overview of the Guide system organization and implementation. Section 3 presents the main design concepts of the Guide language and illustrates them by simple examples. Section 4 is devoted to concurrency and synchronization. Section 5 describes the run-time representation of the main constructs of the language and the principle of the compiler. Section 6 presents some early experience in programming distributed applications. Section 7 summarizes the contribution and the current state of the project.

## 2. SYSTEM OVERVIEW

In this section, we give a brief overview of the global organization of the Guide system, in order to provide the framework for the description of the language. An experimental version of the system has been implemented on top of Unix[1] on workstations interconnected by a local area network. This implementation is described in [Decouchant 88b].

### 2.1. Objects

[1] Unix is a registered trademark of AT&T

An object encapsulates a set of data (the state of the object) and a set of procedures (the operations or methods) that operate on these data. Objects are organized into types (a type specifies an interface common to a set of objects) and classes (a class specifies an implementation common to a set of objects, and a mechanism for creating instances). Types and classes are described in detail in section 3, together with the language constructs that support them.

Objects are persistent, i.e. an object survives the execution of the method in which it was created (temporary objects also exist). Therefore, objects must have an internal name independent of any specific program execution. This naming scheme is provided by system-wide references. A reference is an information (currently 64 bits) that uniquely defines an object. When a method of an object is called, a locating algorithm finds the object (possibly on a remote node). In the body of a method, persistent objects are accessed by means of views. A **view** is a typed variable which is akin to a pointer; it is essentially a container for a reference to a persistent object. Calls to a method of an object may only be performed through a view on that object.

As a consequence of persistence, there is no file system in the usual sense. The two functions of a file system (information storage and naming) are split between the object storage system and a directory service that runs as an application. Object storage is described in section 2.3.

### 2.2. Execution structures

Objects are passive data structures. Execution structures are organized in two levels, called **jobs** and **activities**. A job usually corresponds to the execution of an application. A job is the main support for distributed execution and can be viewed as an object space that may spread over several nodes. Within a job, execution is carried out by an arbitrary number of concurrent activities, i.e. sequential threads of control. An activity executes a sequence of calls to object methods. Objects are dynamically bound within a job as a result of these calls. If a call is made to an object located on a distant node, the job and activity "diffuse" on that node, and a remote method call is performed; however, the distribution is not visible at the language level (except if the location of an object must be made explicit, see section 4). The execution structures are described in figure 1.

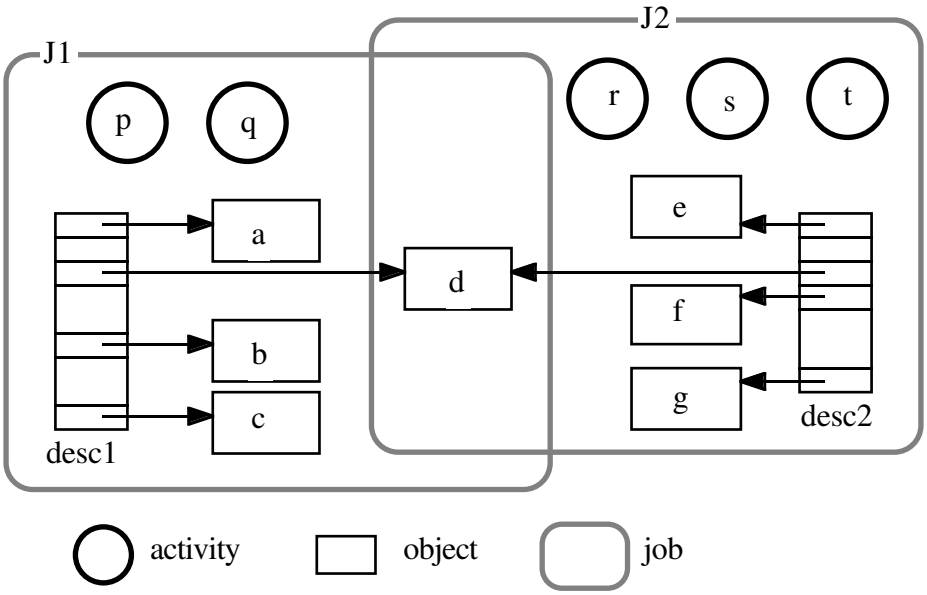


**Figure 1.** Execution structures

Communication between activities, within a job or between different jobs, is performed through shared objects (e.g. object d in Fig.1 is shared between jobs J1 and J2). The usual communication mechanisms such as buffers, message

queues, etc. may be implemented by shared objects. The internal consistency of these shared objects is ensured by synchronization mechanisms, described in section 4, together with the control of concurrent activities within a job.

### 2.3. Object location and storage

Object storage is organized as a two-level hierarchy. At the lower level, a Storage System (SS) is in charge of the long term storage of persistent objects. At the upper level, a Virtual Object Memory (VOM) supports the execution of jobs (i.e. objects bound in a job are addressed in the VOM). Both the SS and the VOM are distributed. When a method of an object is called, a locating algorithm (using a location hint in the reference) determines the current location of the object. If the object is not currently in VOM, an image of its data is loaded from SS and the call is performed. By default, if an object is in VOM, it remains on its current node; if not, it is loaded on the calling node. In a future version, it is intended that the loading policy be adjustable according e.g. to a load leveling algorithm.

## 3. OBJECT MODEL

### 3.1. Types and classes

A **type** is essentially the description of an interface. This interface specifies a behavior that is common to all objects of the given type, in terms of operations applicable to these objects; this is the usual concept of an abstract type. Each operation (or **method**) in a type description is defined by a **signature**, which specifies the name of the method, the types of its parameters, and whether each parameter is an argument (`IN`) or a result (`OUT`). In addition, each method has a specification, which describes its effect. In the current state of the art, the specification is only a comment and is not subject to any formal processing. However, it should be considered an integral part of the type definition.

A type may specify a component of the state of the object (i.e. a variable) as visible. Such a visible variable is also called an **attribute**. If `x` (a variable of type `t`) is an attribute of an object `obj`, two operations are implicitly defined : a function `x-val:t`, such that `obj.x-val` returns the value of `x`, and a procedure `x-set(IN v:t)`, such that `obj.x-set(v)` sets the value of `x` to `v`. As a shorthand, these operations are expressed with the usual notations (e.g. `x:=v` for `obj.x-set(v)`), and the declaration of an attribute `x` of type `t` takes the usual form `x:t`. An initial value may be specified for an attribute. If a declaration (with initial value) is prefixed by `CONST`, then the value of the attribute may not be modified (i.e. only `val` is provided).

We now give two examples of a type definition. The first one defines a document description. The second one describes a library (a collection of documents).

```
TYPE Document_descr IS
        key:    Integer;
        title, author:  String;
        date_borrowed, date_returned : REF Date;
        METHOD Init;
           {set initial values}
        METHOD Consult;
           {display information about the document}
        METHOD Get_text: REF Document;
           {gives access to the text of the document, defined by
            type Document (not specified here)}
END Document_descr.

TYPE Library IS
        METHOD add_document(IN doc: REF Document_descr):Integer;
           {add a document to the library, returns a key}
        METHOD search(IN key:Integer;OUT doc:REF Document_descr);
                SIGNAL not_found;
           {look for a document specified by its key}
        METHOD borrow(IN doc:REF Document_descr);
                SIGNAL is_out;
           {borrow a document from the library}
```

```
        METHOD return(IN doc:REF Document_descr);
           {return a borrowed document to the library}
        METHOD list;
           {print the contents of the library}
        … <other methods, e.g. search by keywords, etc>
        METHOD Init;
           {Initialize the library}
END Library.
```

In the above definition, the keyword **REF** <type> defines a view for an object of type <type> (views were defined in 2.1). The semantics of **REF** parameters in method signatures is essentially call by reference. Call by value is also possible, but is restricted to internal objects (for the time being, internal objects are defined as objects of elementary types, such as `Integer`, `Char`, etc; this definition will be extended in 3.6). The keyword **SIGNAL** specifies that a named exception may be raised under specified conditions.

A **class** specifies a particular implementation of a type. A class definition is common to all instances of the class. It includes a description of the internal state of the object, as a set of **instance variables**, and the program of the methods. The attributes declared in the type definition are automatically included as instance variables.

For instance, two possible implementations of type `Library` are defined by the following class descriptions.

```
CLASS Library_list IMPLEMENTS Library IS
        doc_list: REF List OF REF Document;
        METHOD search(IN key:Integer; OUT doc:REF document);
                          SIGNAL not_found;
           {looks for a document specified by its key}
           BEGIN
             <program of search using list representation>
           END search;
        …
        <program of other methods>
        …
END Library_list.

CLASS Library_array IMPLEMENTS Library IS
        doc_list: REF Array OF REF Document;
        METHOD search(IN key:Integer; OUT doc:REF document);
                          SIGNAL not_found;
           BEGIN
           <program of search using array representation>
           END search;
        …
        <program of other methods>
        …
END Library_array.
```

As usual in object-oriented languages, an instance of a class is created by calling the method `New` of the class. For instance, if `lib` is a view on type `Library`, the statement: `lib:=Library_list.New` creates a new (uninitialized) instance of class `Library_list` and assigns it to `lib`. Thereafter, methods may be invoked on this object by calls like `lib.search(…)`. The reference may be reassigned to another object of type `Library`, possibly with a different representation (e.g. an instance of class `Lib_array`), but the calling sequence is unchanged.

### 3.2. Type conformity

The last example has shown that a view may be assigned objects having different implementations (but of the same type as the view). This facility is equivalent to that provided in Modula-2 [Wirth 85] by opaque types.

Object-oriented languages go one step further by allowing assignment of a variable to objects of *different* types. For instance, a variable in Smalltalk-80 [Goldberg 85] can be assigned any object; however, no type checking is performed, and possible errors (e.g. non existent method) may only be detected at run-time. We want to allow some of the

flexibility of Smalltalk, while performing type checking at compile time as far as possible. This raises the following question: under what conditions may an object be assigned to a view of a different type?

The conditions for such an assignment have been explored [Cardelli 85]. Let `ref` be a view of type `T1`. Then, an object of type `T2` may be safely assigned to `ref` if `T2` conforms to `T1`. "Safely" means that subsequent operations on `ref`, allowed by its type `T1`, will be legal if applied on the object of type `T2`. Conformity is defined as follows:

• For each method defined in the interface of `T1`, there exists a method of the same name in the interface of `T2`.

• Each method of `T2` has the same number of arguments (**`IN`** parameters) and of results (**`OUT`** parameters) as the corresponding method of `T1`.

• For each method `m`, the type of its arguments in `T1` conforms to the type of its arguments in `T2`.

• For each method `m`, the type of its results in `T2` conforms to the type of its results in `T1`.

• Every type conforms to itself.

This definition implies that for each attribute in type `T1`, there must exist an attribute of the same name and type in type `T2`. This agrees with the notion of type extension [Wirth 88], that applies to the special case where types `T1` and `T2` are records (i.e. they only specify attributes, and no explicit methods).

Note that conformity is a purely syntactic relationship that involves no semantic equivalence, since it only relates the signatures of the methods, not the specifications. Conformity has therefore no automatic implications regarding the actual behavior of the objects. However, in the special case of records, the only (implicitly defined) methods are `val` and `set`, whose semantics is well-defined; therefore, in this case, syntactic conformity guarantees consistent behavior.

### 3.3. Subtyping and type restrictions

We now describe two ways of using relations between types to help structuring the applications: subtyping (or type extension) and type restriction. In both cases, conformity is used to ensure a safe use of types.

*3.3.1. Subtyping*. Subtyping is a well-known way of specifying shared behavior between types. Informally, type `T2` is a subtype of type `T1` (and `T1` is a supertype of `T2`) if the interface of `T2` provides at least the same operations (including attributes) as the interface of `T1` (it may provide more operations). The subtyping relationship defines a hierarchy between types. In the current design of Guide, this hierarchy is simple, i.e. a type has only one supertype.

Subtyping allows a conceptual economy : to define type `Book_descr` , we only need to specify the additional attributes and operations to be added to those defined in `Document_descr`. Thus `Book_descr` may be defined as:

```
TYPE Book_descr SUBTYPE OF Document_descr IS
        volume_number: Integer;
        publisher: String;
END Book_descr.
```

This type defines two additional attributes and no additional explicit methods.

Subtyping embodies a "is-a" relationship. In other terms, if `Book_descr` is a subtype of `Document_descr`, we expect an object of type `Book_descr` to behave like an object of type `Document_descr`, i.e. we expect to be able to provide a `Book_descr` wherever a `Document_descr` is expected. Let `doc` be a view of type `Document_descr`. We expect the assignment `doc:=<an object of type Book_descr>` to be valid, for all subsequent uses of view `doc` as the argument for a method call or as a parameter. Type conformity, as defined in the last section, precisely guarantees this property. In the Guide language, we specify, by definition, that a subtype must conform to its supertype. As a consequence, methods may be overloaded in subtypes, provided that the conformity conditions are respected. For instance, we now could give another definition of type `Book_descr` .

```
TYPE Book_descr SUBTYPE OF Document_descr IS
```

```
        volume_number: Integer;
        publisher: String;
        METHOD Get_text: REF Book;
           {gives access to the text of the book}
END Book_descr.
```

Now, the method `Get_text` of the supertype `Document_descr` is overloaded, and the type of the result parameter is changed (we assume that `Book` is a subtype of `Document`). This is legal according to the conformity rule, as the range of a result parameter is narrowed.

Since the conformity relationship may be statically checked, the language is strongly typed. Conformity has been used in at least two object-oriented languages (Emerald [Black 86a] and Trellis/Owl [Schaffert 86]). The use of conformity is not limited to subtyping. Type restriction is another application, which we now describe.

*3.3.1. Type restrictions*. It is often desirable to provide a restricted view on an object. For instance, suppose one wants to disallow the use of some methods of an object of type `T` by a user. One way of achieving this is to provide this user with a view of a type `T1` which is a restriction of `T`, i.e. which only provides a subset of the methods of `T`. `T1` only acts as a "filter" for the use of an object of type `T` and need not be implemented by a class. The above discussion shows that `T` must conform to `T1`.

The following type definition illustrates type restriction. It specifies a restricted interface to type `Library` that may be provided, for instance, to the library staff. This interface is a subset of the interface provided by type `Library`.

```
TYPE Library_Manager IS
        METHOD add_document(IN REF doc:Document_descr):Integer;
           {add a document to the library, returns a key}
        METHOD search(IN key:Integer;OUT doc:REF Document_descr);
                SIGNAL not_found;
           {look for a document specified by its key}
END Library_Manager.
```

The following type definition illustrates conformity for type restrictions. It defines another restriction of type `Library`, which is made available to users. Users are only allowed to borrow books, not other types of documents. Therefore the `borrow` and `return` methods are overloaded and may only accept references to documents of type `Book`. The conformity rule is satisfied.

```
TYPE Library_User IS
        METHOD borrow(IN b:REF Book_descr);
                SIGNAL is_out;
           {borrow a book from the library}
        METHOD return(IN b:REF Book_descr);
           {return a borrowed book to the library}
        METHOD search(IN k: key; OUT REF doc:Document_descr);
                SIGNAL not_found;
           {looks for a document specified by its key}
        END Library_User.
```

Note that conformity and subtyping are really two different concepts, although they have often been confused in the literature. Specifying that a subtype must conform to its supertype in a language is a design decision which results from the requirement of statically checked type safety. However, some language designers have decided otherwise. For example, subtypes in the current version of Eiffel do not have to conform to supertypes. The decision of allowing non-conform subtyping is usually motivated by the restriction imposed by the "contravariant" behavior of method arguments. If conformity is not checked at compile time, illegal operations (e.g. unknown method name, or wrong parameter type) are detected at run time. This is the general case in untyped languages like Smalltalk-80.

The restrictions imposed by conformity may be alleviated by dynamic checking. This is allowed in Guide, by the **TYPECASE** construct:

```
TYPECASE ref OF
```

```
        T1: <sequence of statements>1 END;
        T2: <sequence of statements>2 END;
        …
        OTHER: <sequence of statements>
END;
```

The appropriate statements are executed according to the actual type of the object named by `ref`. The view `ref` is considered as being of type `T1` within `<sequence of statements>`$_1$, etc. If the **OTHER** branch is selected, `ref` is considered as being of its original type (the type of its declaration). This construct is illustrated by the following sequence, using the types defined above.

```
…
desc: REF Document_descr;
user_lib: REF Library_user;
…
user_lib:=<an instance of a class that implements Library>
user_lib.search(key,desc);
TYPECASE desc OF
        Book_descr: user_lib.borrow(desc) END;
        OTHER : <error sequence>
END;
```

This ensures that `user_lib.borrow` is only called if the reference returned by `search` is of type `Book_descr`.

### 3.4. Subclassing and inheritance

The subtyping hierarchy is paralleled by a subclassing hierarchy, which allows sharing of physical properties (i.e. attributes and method implementations) between objects that already share a common behavior. Let `T2` be a subtype of `T1`. If class `C1` implements `T1`, a class `C2` which implements `T2` may be constructed by inheritance from `C1` as follows.

- `C2` inherits all the instance variable definitions present in `C1`.
- Additional instance variables may be defined in `C2`.
- `C2` inherits all the methods defined in `C1`.
- An inherited method may be overloaded in `C2`, as long as the conformity conditions are respected.
- Additional methods may be defined in `C2`, namely those which are part of the interface of `T2` and which are not included in the interface of `T1`.

As usual, when an inherited method is overloaded, the overloaded method is still accessible within the new program (through a pseudo-variable `SUPER`) .

As a conclusion, we may again stress the distinction between type and class, and the parallel relations defined by subtyping and subclassing. Subtyping defines a common *behavior*, in terms of method signatures and specifications, while subclassing (i.e. inheritance) defines a common *implementation*, in terms of data structures and executable code.

### 3.5. Object composition

Two different composition mechanisms are provided. The first one allows to embed *objects* within the state of other objects; these embedded objects are said to be **internal** to the enclosing object and are not visible to the object management system. The second mechanism allows to embed *references to objects* within the state of other objects.

Internal objects extend the notion of instance variables: instance variables may be instances of any class and are not restricted to be instances of predefined classes that implement elementary types (`Integer`, `Boolean`, `Char`, etc). The main motivation for internal objects is efficiency. An internal object is not visible from outside the enclosing object and has no specific representation in secondary storage; it may not be shared between several objects. An internal object

is essentially a component of the state of the enclosing object. In the current implementation, internal objects must have a constant size; this restriction will be removed in future implementations.

The second composition mechanism, composition through references, is much more general. It allows composite structures of arbitrary size to be built from component objects, and allows objects or substructures to be shared. Since references provide a system-wide naming scheme, composite structures, as opposed to individual objects, may be distributed on several nodes (although this distribution is not apparent at the programming language level).

Object composition is usually done through predefined constructors (3.5) such as arrays, records and lists, which are the main building blocks for the construction of complex structures. The body of a document may be defined as a list of chapters, each of which is a list of paragraphs, etc. Several documents may share a chapter (or a sublist of chapters).

Note that the choice of the composition method is made explicitly by the programmer, according to the specific characteristics of the application (for instance a set of small objects which are expected to be often used together is conveniently represented by internal objects). A similar two-level scheme is provided by the AVANCE system [Björnerstedt 88], in the form of "packets" and "datatype values". Emerald also provides the equivalent of internal objects, but the decision to implement an object as internal is made by the compiler. Efficiency (the main motivation for internal objects) is especially crucial in a system based on dynamic binding, the usual case in distributed systems. As large applications are developed, we expect to gain experience on the criteria for the choice of the composition methods. Since such methods have only been in recent use on prototype systems, experimental data are still very scarce.

Object composition and sharing make garbage collection necessary. We do not consider this issue in this paper.

## 4. SUPPORT FOR CONCURRENCY AND DISTRIBUTION

### 4.1. Concurrent activities

The execution of concurrent activities within a job is controlled by the following construct (concurrent clause):

```
COBEGIN
   [<label> : <object invocation>]*
COEND  [<condition>]
```

Each object invocation enclosed between **COBEGIN** and **COEND** is executed as a separate activity. The calling act-ivity (i.e. the activity in which the concurrent clause is called) is suspended. Its resumption is controlled by the termina-tion condition, which has the form of a boolean expression using the labels of the concurrent invocations. Each of these labels is interpreted as a boolean variable that specifies whether the corresponding invocation is terminated. By default, if no termination condition is specified, the calling activity is resumed when all concurrent invocations are terminated.

### 4.2. Synchronization of shared objects

Activities communicate through shared objects. Object sharing must be controlled to guarantee that the shared data stay in a consistent state. Our basic choice is to express synchronization as a set of constraints associated with objects, not as primitives appearing within activities. This is fully consistent with the object approach, since the specification of the synchronization constraints is concentrated in the class that describes the object instead of being spread out in methods that use the object. In addition, this synchronization specification is shared by all instances of the class.

The only way for an activity to access or modify an object is to execute a method of this object. Therefore, we specify synchronization as a set of **activation conditions**, which form a **control clause**. Each activation condition is attached to a method and must be satisfied before the execution of this method may start. If no activation condition is

attached to a method, then the execution of this method is unconstrained. The syntax of the control clause is as follows:

```
CONTROL [<method name> : <activation condition>]*
```

where `<activation condition>` is a boolean expression which may contain the following parameters: instance variables which represent the internal state of an instance; actual parameters of the method; synchronization counters.

Synchronization counters are internal data that specify, for each method of a given object, the total number of invocations, the total number of completed executions, the current number of pending invocations, etc. These counters are automatically updated by the system. Synchronization based on counters was introduced in another context by Robert and Verjus [Robert 77]. The following counters are defined for each method `m`:

| | | |
|---|---|---|
| `invoked(m)` | : | number of invocations of method `m` |
| `started(m)` | : | number of accepted (non-blocked) invocations of method `m` |
| `completed(m)` | : | number of completed executions of method `m` |
| `current(m)` | : | number of activities currently executing method `m` |
| `pending(m)` | : | number of activities currently blocked on invocations to `m` |

The following relationships hold :

```
current(m)    = started(m) - completed(m)
pending(m)    = invoked(m) - started(m)
```

The first three counters are created and initialized to 0 when the instance is created. They record total numbers, and they are non-decreasing.

Synchronization constraints are illustrated by the following program, which implements a bounded buffer used for communication between activities. The type `ProducerConsumer` is supposed to have been introduced elsewhere:

```
CLASS FixedSizeBuffer IMPLEMENTS ProducerConsumer IS
        CONST size=<some constant>;
        buffer : ARRAY[0..size-1] OF Element;
        first, last: Integer = 0, 0;
      METHOD Put(IN m: Element);
        BEGIN
        buffer[last]:=m;
        last:=last+1 MOD size;
        END Put;
      METHOD Get(OUT m: Element);
        BEGIN
        m:=buffer[first];
        first:= first +1 MOD size;
        END Get;
      CONTROL
        Put: (completed(Put) - completed (Get)<size)
              AND current(Put)=0;
        Get: (completed (Put) > completed (Get))
              AND current(Get)=0;
END FixedSizeBuffer.
```

Special shorthand forms are used in the most usual cases. For example, mutual exclusion between all methods in an object is expressed by a single activation condition which is common to all methods:

$$\Sigma\ \mathtt{current}(m_i) = 0$$

Since this condition is frequently used, it is replaced by a single keyword **`EXCLUSIVE`** in the control clause.

It should be noted that activation conditions are expressed using only boolean expressions. Modification of internal variables or more general algorithms are not allowed. As a consequence, some synchronization schemes can not be

directly expressed, and must involve the use of additional methods. The implementation of this synchronization scheme is described in [Decouchant 88a].

### 4.3. Distribution aspects

Distribution is managed by the underlying system and is mostly invisible at the programming language level. Decisions related to distribution are made at the following times:

a) At object creation, the location of the object must be specified. More precisely, the Secondary Storage system is organized in units called containers; the creation primitive specifies the initial container; a mapping of containers to physical nodes is specified separately. By default, the object is created on the node where the call takes place.

b) At method invocation, if the object is not in VOM, the node where it should be loaded must be specified. By default, the object is loaded on the calling node. In the future, this will be governed by a load-balancing policy.

## 5. IMPLEMENTATION

### 5.1. Run-time representation of objects

The representation of an object (an instance of a class) is kept in the Guide system as two separate parts: the state image and the class image. The state imagecontains its instance variables (i.e. the representation of the state of the object. The class image contains the representation of the methods (executable code and linkage information). The class image is shared by all instances of the class. The run time representation of a view is a descriptor that contains a reference to the state image of the object currently pointed by the view; this state image in turn contains a reference to the corresponding class image. In addition, when the object is loaded in virtual memory on a node, the view points to descriptors for its state and class images in a local object table, as shown on Figure 2 (if the object is currently loaded in virtual memory on a remote node, a remote method call protocol is used).

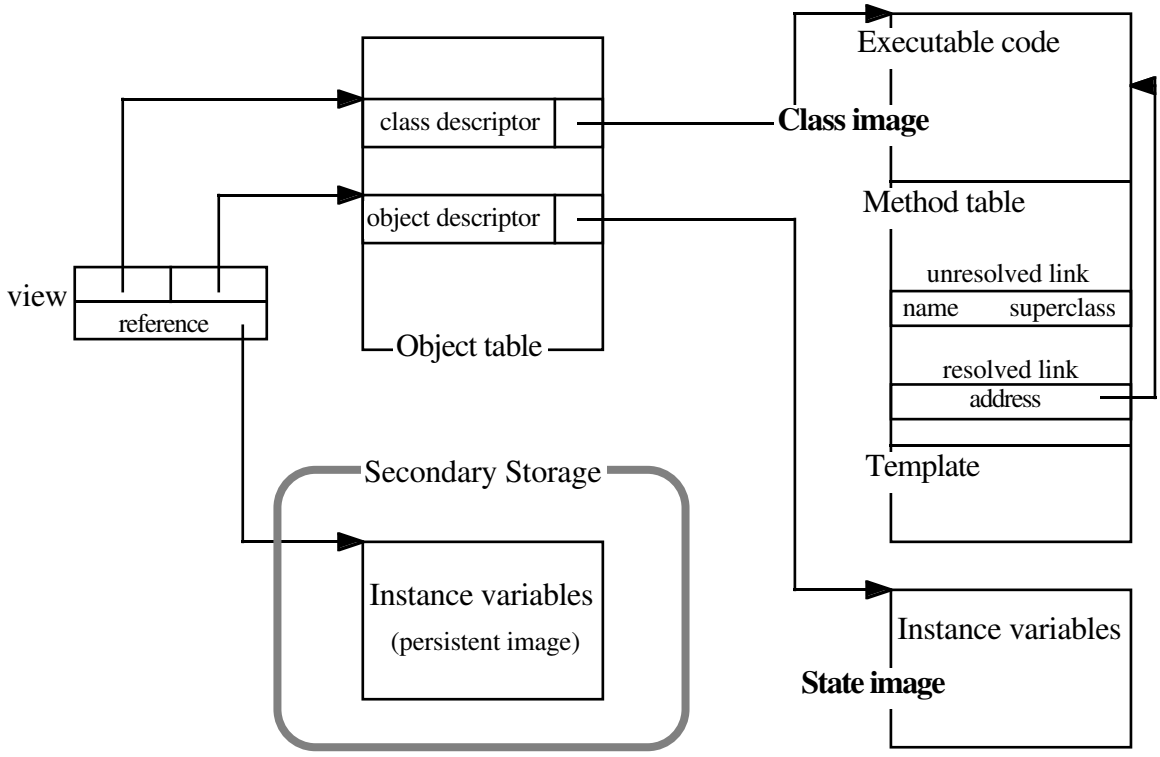


**Figure 2**. Run-time representation of an object

The class image of an object contains three parts: the executable code of the methods; the method table; and a template that represents the internal organization of the state image. This template is used whenever one needs to find the references embedded in an object, e.g. at object creation or for deep copy or garbage collection. The method table contains an entry for each method that may be invoked on the object . If the method is defined in the class (i.e. not inherited), the entry contains its address. If the method is inherited, the entry initially contains un unresolved link, which consists of the method name and the identification of the ancestor class from which the method is inherited. This link is resolved at the first call and subsequently replaced by the address of the method.

### 5.2. Principle of the compiler

The input to the Guide compiler is the source programs of types and classes. From the source program of a type, the compiler generates a type description. From the source program of a class, the compiler generates a class description and an executable class object. Type and class descriptions are stored in an internal form by a type and class manager. These descriptions are used by the compiler for conformity checking.

The current version of the compiler generates C code for the methods defined in classes. The overall translation scheme is organized in 5 passes. Pass 1 performs lexical syntactic parsing, using the `lex` and `yacc` standard tools, and generates a syntax tree. Pass 2 performs conformity checking and semantic analysis, using the descriptions of types and classes used by the type or class being compiled; it also generates a description for this type and class. Pass 3 to 5 only apply to classes. Pass 3 generates C code for the methods. Pass 4 translates this code using the C compiler (`cc`). Pass 5 creates the class object. It generates the method table and template for the class. Finally, it stores the object class as a persistent object in secondary storage and registers its name with the name server. The object class may thereafter be used by calling one of its methods (`New` is usually called in the first place to create an instance).

In order to start an application, an initial method and an initial object must be provided. The application is started within a job by an activity which calls this method. Thereafter, other objects are dynamically linked when needed, and the job possibly spreads out to distant nodes. Currently, the only way to integrate an existing program in a Guide application is to "encapsulate" it into a Guide object and to provide an interface to it in the form of a Guide type. This has been done, for instance, with the X-Window server [Scheifler 86], which can now be used by any Guide application.

## 6. EXPERIENCE

At the time of writing, the Guide language has been in experimental use for about six months. Programs in Guide have been run on the first version of the Guide operating system, totaling about 7000 lines of source code. We report here some preliminary results of this experience.

A typical example of an application program written in Guide is a simple electronic mail system. Mail is regarded as a typical distributed application and is often used as an illustration for distributed systems structuring (e.g. [Almes 84]). In its simplest form, a mail service provides the following functions to the registered clients: send message; display message list; read message; delete message. The administrator of the mail service may display the list of registered clients, add a new client to this list, and remove a client from this list.

Programming this application in Guide involves a structure which differs from that of most traditional mail service implementations. A salient feature of this structure is that distribution never explicitly appears, which results from the basic decision of providing location transparency at the system level. Actually, the objects used in the mail service (lists of clients, mailboxes, messages, etc) may be distributed on several nodes, but this distribution is not imposed by the structure of the mail program (e.g. the messages contained in a mailbox need not be on the same node, etc). The distribution of the objects may be governed by considerations of efficiency, security, availability, performance, etc.

The application is organized around the following object types:

• messages (Type `Message`). Objects of this type are the messages that are sent and received by clients.

• mailboxes (Type `Mailbox`). Objects of this type are essentially implemented as lists of references to messages.

• directories (Type `Mail_Directory`). Objects of this type implement the mapping between a list of client names and the mailboxes associated with them.

• mail service (Type `Mailer`). Objects of this type implement the functions provided to clients of the mail service.

When a client calls the mail service, a mailer object (an instance of type `Mailer`) is created and lives until the end of this client's session. This object is created on the node where the mail service was called. Other objects such as mailboxes, messages, etc, may be created on other nodes. Clients are not aware of the distribution of these objects.

Figure 3 shows the structure of the mail service. Rectangles with round corners represent collections of instances of classes that implement the above types; these instances are shown as ordinary rectangles. Arrows represent references to objects. Distribution does not explicitly appear on this diagram, since references are used to find objects irrespective of their actual location. Instances of a class that are currently loaded on a node share the program of the class on that node.

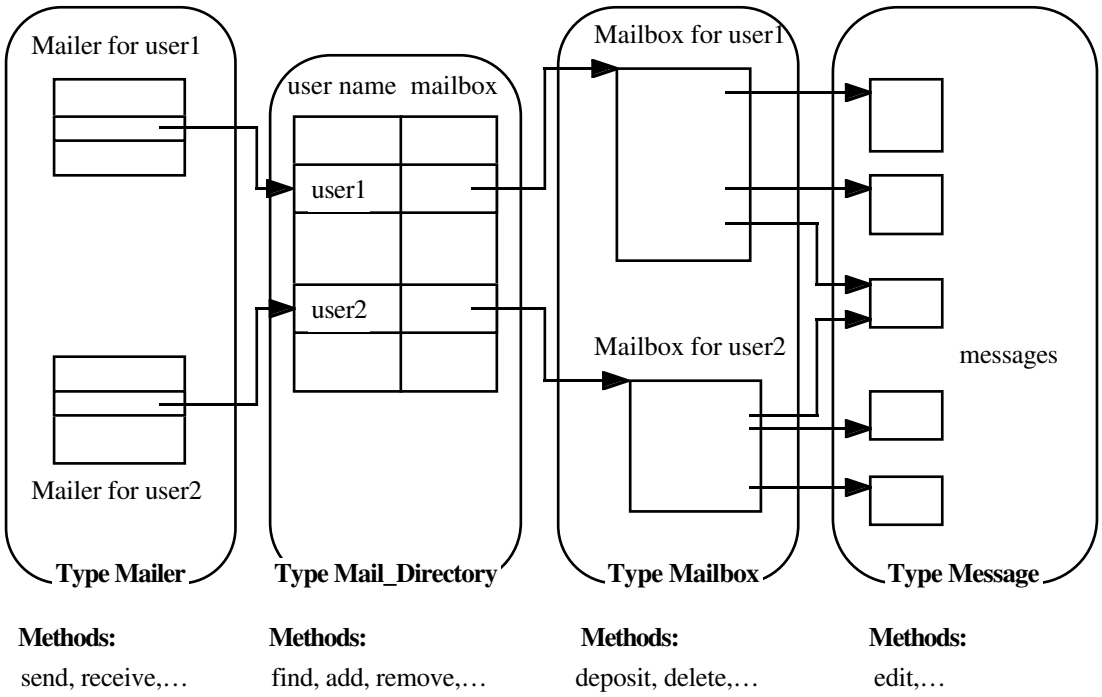


**Figure 3**. Structure of the Mail application

The main lessons learned from the preliminary experience with the system may be summarized as follows.

• *Type and class model*. The separate definition of types and classes was found an extremely useful feature. It provides the power of abstraction (separation between type and representation) while keeping the flexibility of an object model. This works in both ways: provision of multiple implementations (classes) for a same interface allows easy and transparent implementation modifications, while provision of multiple interfaces for a same class, via type restriction, is a very natural way of filtering access to a shared data structure. For instance, the user and administrator interfaces to the mail service are easily implemented by using type restriction on the type `Mail_Directory` (users may only use `find`, while administrators may also use `add` and `remove`).

*Persistent objects*. Using persistent objects frees the programmer of the burden of explicit save and restore operations on files. Indeed, the notion of a file system entirely disappears. This freedom has a price in terms of performance. Using internal objects greatly improves performance at the expense of flexibility, since this amounts to a form of static linking of part of an application. A development method that we found convenient is to develop and debug an application using fine-grained persistent objects, and to restructure it to use internal objects for small, clustered units (such, for example, as directory entries in the mail application) once the code is stable. Such restructuring only involves a moderate amount of change.

*Location transparency*. Location transparency is ensured by the object management system, and object locations are usually invisible to the application programmer. By default, the actual location of the object is controlled by the system, according to an independently defined policy. There is no distinction, at the language level, between local and remote method calls. As a consequence, it is easy to develop and debug an application on a single node and to transfer it to the network without change. The mail service application was developed in this way. However, explicit control on object location may still be applied if needed, for example to implement server applications in which the service is to be provided by a specified node or set of nodes.

## 7. CONCLUSION

We have designed and implemented a language for the programming of distributed applications, closely integrated with a distributed operating system that supports persistent objects. This work is part of an ongoing investigation in integrating operating systems and programming language techniques to support objetc-based distributed applications.

A prototype version of the language and the system is operational on a local area network of Suns and Bull SPS-7 workstations. The first experience with this prototype has convinced us that:

• The safety provided by strong typing and compile time checking can be combined with the flexibility allowed by dynamic binding and subtyping.

• The separation between type and class combines the advantages of data abstraction (separation between interface and implementation, type restriction) and those of inheritance (conceptual and physical economy).

• Location transparency can be provided for persistent objects supported by the language, both for permanent storage and for object invocation; specific object location is still available if needed.

• A synchronization mechanism fully compatible with the object model can be designed.

Further work aims to extend the functions of the system and to improve its performance. Experiments with real-life applications (cooperative work on large structured documents) will provide a testbed for the language and system.

**Acknowledgments.** All members of the Guide project have contributed to this work. D. Decouchant and X. Rousset de Pina participated in the design and implementation of the synchronization mechanism and the run-time support. Many ideas evolved through discussions within the Comandos project, especially with our colleagues from Trinity College, Dublin. This work was partially supported by the Commission of European Communities through ESPRIT project 834 (Comandos). This work was also partially supported by Centre National de la Recherche Scientifique and Centre National d'Etudes des Télécommunications.

## References

[Albano 85]
Albano A., Cardelli L., Orsini R., Galileo: a strongly typed, interactive conceptual language, *ACM Trans. on Database Systems*, vol. 10, 2 (june 1985)

[Almes 84]
Almes G., Black A., Bunje C., Wiebe D., Edmas: a locally distributed mail system, *Proc. 7th Int. Conf. on Software Engineering,* March 1984, pp. 55-66

[Alves Marques 88]

Alves Marques J., Balter R., Cahill V., Guedes P., Harris N., Horn C., Krakowiak S., Kramer A., Slattery J., Vandôme G., Implementing the Comandos architecture, *Proc. 5th Annual ESPRIT Conference*, Bruxelles, North-Holland (nov. 1988), pp. 1140-1157

[Bal 88a]
Bal H.E., Steiner J.G., Tanenbaum A.S., Programming Languages for Distributed Systems, *Vrije Universiteit Amsterdam*, rapport IR-147 (feb. 1988)

[Bal 88b]
Bal H.E., Tanenbaum A.S., Distributed programming with shared data, *Proc. IEEE CS 1988 Int. Conf. on Computer Languages*, Miami, Fl., pp. 82-91 (oct. 1988)

[Bancilhon 87]
Bancilhon F., Briggs T., Khoshafian S., Valduriez P., FAD: A powerful and simple database language, *Proc. 13th VLDB Conf.*, Brighton (sept. 1987)

[Björnerstedt 88]
Björnerstedt A., Britts S., AVANCE: an object management system, *Proc. ACM Conf. on Object-Oriented Progrramming Systems, Languages and Applications*, San Diego, SIGPLAN Notices, 23, 11 (nov. 1988), pp. 206-221

[Black 85a]
Black A.P., The Eden programming language, *Tech. Report 85-09-01, Univ. of Washington* (sep. 1985)

[Black 85b]
Black A.P., Supporting distributed applications : experience with Eden, *Proc. 10th ACM Symp. on Operating Systems Principles*, *SIGOPS Operating Systems Review*, vol. 19, 5 (dec. 1985), pp. 181-193

[Black 86a]
Black A.P., Hutchinson N., Jul E., Levy H., Carter L., Distribution and abstract types in Emerald, *IEEE Trans. on Software Engineering*, vol. SE-12 (dec. 1986)

[Black 86b]
Black A.P., Hutchinson N., Jul E., Levy H., Object structure in the Emerald system, *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, Portland, Oregon, SIGPLAN Notices, vol. 21, 11 (nov. 1986), pp. 78-86

[Campbell 74]
Campbell R.H., Habermann A.N., The specification of process synchronization by path expressions, *Operating Systems*, Lecture Notes in Computer Science, Springer-Verlag (1974), pp. 89-102

[Cardelli 85]
Cardelli L., Wegner P., On understanding types, data abstraction, and polymorphism, *ACM Computing Surveys*, vol. 17, 4 (dec. 1985), pp. 471-522

[Comandos 88]
Comandos Project Report: Oscar: preliminary programming language manual, Document D1-T3.2.3.2-880331, ESPRIT Project 834

[Decouchant 88a]
Decouchant D., Krakowiak S., Meysembourg M., Riveill M., Rousset de Pina X., A synchronization mechanism for typed objects in a distributed system, *Workshop on Object-Oriented Concurrent Programming*, San Diego (sept. 1988)

[Decouchant 88b]
Decouchant D., Duda A., Freyssinet A., Paire E., Riveill M., Rousset de Pina X., Vandôme G., Guide: an implementation of the Comandos object-oriented architecture on Unix, *Proc. EUUG Aut. Conf.*, Lisbon (oct. 1988)

[Goldberg 83]
Goldberg A., *Smalltalk-80: The language and its implementation* ; Addison-Wesley Publishing Compagny, [1984].

[Horn 87]
Horn C.J., Krakowiak S., An object-oriented architecture for distributed office systems, *Proc. 4th Annual ESPRIT Conf.*, Brussels (sept. 1987)

[Jul 88]
Jul E., Levy H., Hutchinson N., Black A., Fine-grained mobility in the Emerald system, *ACM Trans. on Computer Systems*, vol. 6, 1 (feb. 1988), pp. 109-133

[LeBlanc 85]
LeBlanc R.J., Wilkes C.T., Systems programming with objects and actions, *Proc. 5th Int. Conf. on Distributed Computing Systems*, Denver (may 1985), pp. 132-139

[Meyer 88a]
Meyer B., Genericity versus Inheritance, *Journal of Pascal, Ada & Modula-2*, vol. 7, 2 (march-april 1988), pp. 13-30 [revised version of a paper in OOPSLA'86, *Sigplan Notices*, 21, 11, pp. 391-405]

[Meyer 88b]
Meyer B., *Object-oriented Software Construction*, Prentice-Hall (1988)

[Mullender 87]
Mullender S.J., ed. The Amoeba distributed operating system: selected papers 1984-1987, *CWI Tract 41*, Amsterdam, 1987

[Robert 77]

Robert P., Verjus J.-P., Toward autonomous descriptions of synchronization modules, *Proc IFIP. Congress* (B. Gilchrist, ed.), North-Holland (1977), pp. 981-986

[Schaffert 86]
Schaffert C., Cooper T., Bullis B., Killian M., Wilpolt C., An introduction to Trellis/Owl, *Proc. First ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, Portland, Oregon, SIGPLAN Notices, 21, 11 (nov. 1986)

[Scheifler 86]
Scheifler R.W., Gettys J., The X Window system, *ACM Trans. on Graphics*, vol. 5, 2 (apr. 1986), pp. 79-109

[Tanenbaum 86]
Tanenbaum A.S., Mullender S.J., The design of a capability-based distributed operating system, *The Computer Journal*, vol. 29, 4 (1986) pp. 289-300

[Wilkes 86]
Wilkes C.T., LeBlanc R.J., Rationale for the design of Æolus: a systems programming language for an action/objects system, *Proc. IEEE CS Int. Conf. on Computer Languages*, Miami (oct. 1986), pp. 107-122

[Wirth 85]
Wirth N., *Programming in Modula-2,* Springer-Verlag, 1982 ; 3rd ed. (1985)

[Wirth 88]
Wirth N., Type Extensions, *ACM Trans. on Programming Languages and Systems*, 10, 2 (apr. 1988), pp. 204-214