The Guide Language

ROLAND BALTER*, SERGE LACOURTE* AND MICHEL RIVEILL[†]

*Bull-IMAG/systèmes, 2 rue de Vignate, 38610 Gières, France

[†]Université de Savoie, LGIS, Campus Scientifique, 73370 Le Bourget du Lac, France

The Guide language is an object-oriented language dedicated for programming distributed applications on top of the Comandos platform. This language faithfully reflects all the capabilities of the Comandos model. It combines the facilities of a strongly typed object model with a powerful computational model. Key features of the language are: separation between types and classes, conformant subtyping, distribution transparency, concurrent activities, synchronization mechanism for shared objects, and exception handling. This paper presents the design choices for the Guide language and discusses programming experience gained from the use of the language for the construction of distributed applications.

Received April, 1994

1. INTRODUCTION

Comandos (Construction and Management of Distributed Open Systems) is a co-operative project under the ESPRIT European program aiming at providing an *integrated development environment for distributed applications* (Cahill *et al.*, 1993). To achieve this goal, Comandos provides a conceptual model of a distributed environment, encompassing both computation and data management. This model presents the functionality of the Comandos platform to application programmers and system administrators. The model is abstract in the sense that it does not require the use of any programming language. Technically it consists of two major components:

- A *computational model*, which allows distributed programs to be defined. This model provides the application designer with a distributed multi-processor virtual machine, in which parallelism is apparent and distribution is hidden.
- A common and extensible *type and data model*. The type model captures the type systems of various object-oriented programming languages and allows interoperability between them. The data model provides abstractions for modelling collections of objects, relationships between collections and classification structures.

Both to facilitate the use of the Comandos model and to encourage its adoption, the model is provided to application designers through one or more *programming languages*. The uniformity of the Comandos model results in programming languages in which a uniform treatment of both transient and persistent data; of both passive and active entities; of both local and remote services are *all* potentially available.

This strategy differs from more classical approaches in which tight coupling of a programming language with support for persistence and distribution is *not* available. Typically, in the classical approach, a set of languages is used for building programs where one language is used to describe persistent types and a further different language is used for interface definition and communication.

The provision of programming language support that integrates all these features is a key aspect of the Comandos project. It is obvious that the features of the model are particularly exploitable by object-oriented programming languages. Two different approaches to the provision of programming language support were considered in the framework of the project: the use of existing languages (i.e. C++ and Eiffel) and the *definition of a new language environment.*

The first approach allows existing programming environments to be exploited and existing code to be reused when appropriate. The choice of C++ and Eiffel was motivated by their industrial and academic success. However, because of the constraints imposed by using (or modifying) existing compilers, some features of the Comandos model cannot be fully integrated within an existing language. In contrast, defining a new language allows full access to the facilities of the Comandos virtual machine. Both approaches are, however, complementary in that objects written in various programming languages can be combined within a given application.

This paper describes the key features of **Guide**, an object-oriented language designed for the programming of distributed applications on top of the Comandos platform. The run-time support of this language is provided by an object-oriented distributed operating system (the Guide system). The Guide system and language implement most of the distributed object-oriented architecture defined by Comandos. Guide was designed and implemented as a joint project between University of Grenoble and Bull Research Centre (Balter *et al.*, 1991; Freyssinet *et al.*, 1991).

Object-oriented languages have attracted much attention in 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. A major design decision of Guide was a close *integration* of the different aspects of the system: programming language, execution structures and long-term storage of information. This integration has been achieved through the uniform use of the object paradigm. As a matter of fact the object in Guide is the unit of modelling and programming, the unit of sharing, the unit of protection, and the unit of storage.

This paper is organized as follows. Section 2 gives an overview of the Guide computation model. Then, Sections 3, 4 and 5 describe three key features of the Guide language, i.e. separation between types and classes, synchronization through shared objects, and exception handling. Section 6 presents an overall evaluation of the language, based on a number of experiments carried out both inside and outside of the project. Finally, section 7 describes the current status and the future of this work.

2. OVERVIEW OF THE GUIDE COMPUTATIONAL MODEL

As stated above, the Guide language has been designed to faithfully reflect all the capabilities of the Comandos computational model. The main abstractions provided by this model are summarized in this section.

2.1. Object model

Objects are the units of modelling and programming. An object encapsulates a set of data, the state of the object and a set of operations, or methods, which operate on this data. Every object is an instance of a *type*, where the type describes the behaviour of this object. Every object also possesses a *class* which describes an implementation of a type, that is the representation of that object and the code of its methods. The properties of types and classes, as well as their relationship are detailed in section 3.

To create an object, a generic *New* operation is provided by all classes (classes are first-class objects). This operation creates a new object and returns a *reference*. Thereafter, methods may be invoked on this object by calls using this reference. A reference contains object identity information (i.e. a system-wide unique identifier) and possibly location hints.

Objects are accessed through *variables*. Variables may be considered as containers for object references. More than one variable may be used to access a given object. A variable possesses a type; when accessing an object through a variable, only the interface described by the type of that variable may be used.

The construction of complex structured objects is achieved by embedding references to objects within the state of other objects. This mechanism allows composite structures of arbitrary size to be built from component objects, and allows objects or sub-structures to be shared. Since references provide a system-wide naming scheme, composite structures, as opposed to individual objects, may be distributed on several nodes.

Object composition is usually done through predefined constructors such as arrays, records and lists, which are the main building blocks for the construction of complex structures. For example the body of a document object 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).

Distribution is, by default, transparent. From the programmer's point of view, remote and local invocations are indistinguishable. In particular, the public attributes of remote objects can be accessed in the same way as for local objects. However, the language also provides constructs to force the creation or execution of an object to take place at a specified node.

Persistence is usually provided in languages for databases [FAD (Bancilhon *et al.*, 1987), Galileo (Albano *et al.*, 1985)] and distributed systems [Emerald (Black *et al.*, 1986)]. Other languages [Trellis/Owl (Schaffert *et al.*, 1985), Eiffel (Meyer, 1992)] provide persistent objects as an add-on feature. Objects in Guide are potentially *persistent*. However, only objects which are reachable from a persistent root are made persistent by the run-time system; other objects are subject to garbage collection. It would be conceptually attractive to treat all objects as persistent in order to define a uniform object model (as for instance in Smalltalk). However, the management of persistent objects involves an overhead that would not be admissible for small objects such as integers, strings, etc. Therefore, Guide provides two other kinds of objects that are managed in different ways: *internal* objects, which are parts of another object's state, and *transient* (or run-time) objects, which are allocated on the execution stack. These objects are introduced to allow programmers to explicitly control the use of system resources according to the requirements of applications. Internal and transient objects are defined by specific declarations in the Guide programming language.

2.2. Execution model

An important design decision is how to relate objects to execution structures. Two main orientations may be taken: (i) to associate execution structures with objects, i.e. to define active objects, each object containing a fixed or variable number of processes, and (ii) to separate objects from execution structures, i.e. to define passive objects being executed by independently defined processes.

We did not find strong logical arguments in favour of either solution. Both have been adopted by existing object-based systems [e.g. active objects in Argus (Liskov, 1985), Eden (Black, 1985) and Emerald; passive objects in Clouds (Dasgupta *et al.*, 1990), Amoeba (Mullender *et al.*, 1990), and SOS (Shapiro *et al.*, 1989)]. The choice is mostly influenced by considerations of efficiency and adequacy for the hardware and

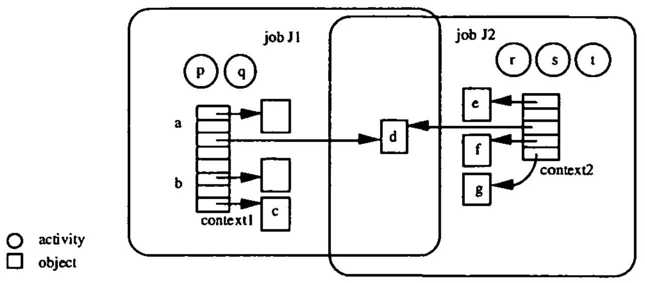

FIGURE 1.

application domain. The applications that we contemplate involve creating many (usually small) objects and building large compound structures out of object components; the parallelism is likely to be coarse-grained. An active object model may be conceptually simpler, since a single abstraction encompasses the two concepts of processes and data structures. On the other hand, the cost of creation and management is higher for active objects and object composition leads to nested virtual memory structures. This led us to adopt a *passive* object model, in order to reduce the cost of object creation and to avoid the complexity of managing compound active objects.

In order to specify the relations between objects and execution structures, we defined a multiprocessor virtual machine called a *job*. A job represents the processing (possibly in parallel) of objects and consists of one or more sequential threads of control, called *activities*. A job is created by an activity of another job to invoke a specified method on some object. A new job initially consists of a single activity.

The execution of an activity consists of nested invocations of methods on objects. Each invocation may take place on any node in the system. Objects (i.e. their code and internal data) are dynamically bound in the virtual address space of the current activity. Thus a job may also be viewed as an addressing window on the global object space, by mapping a set of objects which are shared by the activities. The set of objects mapped within a job is called its context. The composition of the context may dynamically change as objects are mapped or unmapped. See Figure 1.

The concepts of job and activity are similar, respectively, to those of task and thread in the Mach system (Accetta *et al.*, 1986); however, an important difference is that we do not specify an association between a job and a node. Jobs and activities may span several physical nodes; actually, they may dynamically extend themselves or shrink, according to the pattern of object invocations. Distribution is basically hidden from the user of a job.

There is no explicit communication through message passing between activities. Communication and synchronisation between activities (within the same job or belonging to different jobs) is achieved through the *sharing of objects*. The Guide language provides powerful and flexible capabilities for the expression and control of shared objects. This issue is detailed in Section 4.

In addition to the above mentioned features, which are part of the basic Comandos model, Guide also provides additional capabilities for exception handling and for protection. Exception handling is described in Section 5. The protection model is not addressed in this paper.

3. SEPARATION BETWEEN TYPES AND CLASSES

The introduction of strong typing in object-oriented languages is recent. Examples of typed object-oriented languages are Emerald, Trellis/Owl and Eiffel. 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 (in the form of 'abstract' and 'concrete' types).

3.1. Types

A **type** is essentially the description of an interface. This interface specifies a behaviour 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). 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**.

The following example defines a document description:

```
TYPE Document_descri IS
  key: Integer;
  title, author: String;
  date_borrowed, date_returned : REF Date;
  METHOD Init;                  //set initial
    // values
  METHOD Consult;              //display infor-
    //mation 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.
```

In the above definition, the keyword REF <type> defines a reference for an object of type <type>. 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.). The keyword SIGNALS specifies that a named exception may be raised under specified conditions.

3.1.1. Subtyping

Subtyping is a well-known way of specifying shared behaviour between types. Informally, type T2 is a subtype of type T1 (and T1 is a super type 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 super type.

Thus Book_descr may be defined as:

```
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 this type defines two
  //additional attributes, no additional explicit
  //method, but overloads Get_text method
END Book_descr.
```

3.1.2. *Type conformance*

Object-oriented languages go one step further than modular languages by allowing assignment of a variable to objects of *different* types. For instance, a variable in Smalltalk-80 [Goldberg, 1985] can be assigned any object; however, no static 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.

The conditions for such an assignment have been explored (Cardelli and Wegner, 1985). 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.
- For each method m, the signal s in T2 exists 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, 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.

In the Guide language, we specify, by definition, that a subtype must conform to its super type. As a consequence, methods may be overloaded in subtypes, provided that the conformity conditions are respected. The method Get_text of the super type 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 and Trellis/Owl). In Guide, the use of conformity is not limited to subtyping.

3.2. Classes

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, a possible implementation of type Library is defined as follows:

```
CLASS Library_list IMPLEMENTS Library IS
  doc_list: REF List OF REF Document;
  METHOD search (IN key: Integer; OUT doc:REF
    document);SIGNALS not_found;
      BEGIN
       <program of search using list represen-
        tation>
      END search;
  END Library_list.
```

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 variable of type Library, the statement: lib:=Library_list.New creates a new 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.

3.2.1. *Inheritance*

The subclassing hierarchy allows sharing of physical properties (i.e. attributes and method implementations). 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 and additional instance variables may be defined in C2.
- C2 inherits all the methods defined in C1 and an inherited method may be overloaded in C2. Additional methods may be defined in C2, i.e. 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).

3.3. Type/class relationship

In the Guide language as in modular languages, it has been decided to separate the interfaces of abstract structures which represent the types and their implementations which represent the classes. Modular design is enhanced by this approach since information about implementation details remains hidden to the programmer, who only accesses the features defined in the types. The type/class separation was an innovative concept when Guide was designed. Today it is recognised that this approach provides substantial benefits (Lalonde, 1991).

Subtyping is quite different from subclass inheritance. The subtyping relationship defines hierarchy between types. Informally, type T2 is a subtype of type T1 if T2 specializes the behaviour of T1 (T2 provides at least the same methods as the interface of T1). Subclassing defines a common implementation, in terms of data structures and executable code. It is expected that there will usually be a one-to-one correspondence between types and classes (a type having exactly one implementation) and in most cases the class hierarchy copies exactly the type hierarchy. However, we give below several examples where it makes sense to construct the two hierarchies differently.

3.3.1. Types without implementation

Types can be used as *access filters*, as shown in the following example, where *ChanIn* and *ChanOut* implement restrictions on the use of the type *Chan*.

```
TYPE ChanIn IS      TYPE Chan IS     TYPE ChanOut IS
  METHOD input;       METHOD input;    METHOD output;
END ChanIn.           METHOD output;
                    END Chan.

                    CLASS TTY
                    IMPLEMENTS Chan IS
                    //implementation
                    END Chan.

  canal: REF ChanIn;
  canal:= TTY.New // canal is implemented by class
                     TTY
                  // and TTY's type Chan conforms
                     to ChanIn
  canal.input;    // valid instruction
  canal.output;   // illegal instruction
                  // output is not part of the
                     ChanIn type
```

3.3.2. Types with more than one implementation

Using several implementations for a type can be used for handling heterogeneity, as shown below:

```
TYPE Window IS
  READ height: Integer = 0;
  READ width: Integer = 0;
  METHOD resize (In h,w: Integer);
END Window;
```

```
CLASS MyWindow
IMPLEMENTS Window IS
  CONST hmax: Integer = 600;
  CONST wmax: Integer = 400;

  METHOD resize (IN h, w: INTEGER);
  BEGIN
    IF (h<=hmax) THEN
      height := h;
    END;
    IF (w<=wmax) THEN
      width := w;
    END
  END resize;
END MyWindow.

CLASS YourWindow
IMPLEMENTS Window IS
  CONST hmax: Integer = 500;
  CONST wmax: Integer = 500;

  METHOD resize (IN h, w: INTEGER);
  BEGIN
    IF (height+h<=hmax) THEN
      height := height + h;
    END;
    IF (width+w<=wmax) THEN
      width := width + w;
    END;
  END resize;
END YourWindow.
```

A *Window* variable can then take one of the following forms:

```
window: REF Window;
window := MyWindow.New;
          // or window := YourWindow.New;
```

3.3.3. Inheritance versus subtyping

When the type of the subclass conforms to the type of the superclass, the interface of the subclass *contains* the interface of the superclass and instances of a subclass can be used wherever instances of the superclass are expected.

The notion of *containment* may be viewed as similar to that of *conformity* (Lalond, 1991). However, it is sometimes desirable that the interface of a subclass is not strictly a superset of that of its superclass. An interesting design facility consists in inheriting some methods from the parent class and restricting other methods (the code of these methods is overridden). An example of restriction is when arguments of a method in the subclass must be a subtype of the type of the corresponding arguments in the parent class. As a consequence, the subclass interface does not contain the interface of the parent class, and the subtype does not conform to its supertype. This is a typical case where *inheritance is not subtyping.*

A typical example of this usage is when a method must take an argument that is of the same type as SELF, as shown below. In this example, the type *Chapter* does not conform to the type *Document*, although the class *Chapter* inherits from the class *Document*.

```
    TYPE Document IS
      METHOD Concat (IN REF Document);
      ...
    END Document.
  CLASS Document
  IMPLEMENTS Document IS
    l : List OF REF Top;
    METHOD Concat (IN w: REF Document);
    BEGIN
      l.Append(w);
    END Concat;
    ...
  END Document.
  TYPE Chapter IS
    METHOD Concat (IN REF Chapter);
    METHOD Modify;
    ...
  END Chapter.
  CLASS Chapter
  SUBCLASS OF Document
  IMPLEMENTS Chapter IS
    // inherit all the characteristic
    // of Document in order to reuse
    // some algorithms
    // Override Concat
    METHOD Concat (IN c: REF Chapter);
    BEGIN
      c.Modify
      l.Append(c);
    END Concat;
    METHOD Modify;
    ...
  END Chapter.
```

The *Concat* method of the class *Chapter* overrides the method inherited from the class *Document* while at the same time introducing an additional restriction: a chapter can only have children which are themselves chapters. One might want to do this because insert will invoke another method (like Modify) on each of the inserted children.

The type *Chapter* does not conform to the type *Document*. This is because chapters allow other chapters to be inserted as children, whereas documents only allow other documents. A chapter cannot be passed to any arbitrary piece of code that expects to receive a document because it may try to add a child document to it that is a document rather than a chapter.

Inheritance is not subtyping and a class exports only methods defined in its type: a class *Chapter* exports only the methods defined in the type *Chapter* and not all the methods defined in the class *Document*.

4. SYNCHRONIZATION OF SHARED OBJECTS

4.1. Rationale and objective

Many synchronization tools have been proposed for modular or distributed environments (an overview is available in Grass and Campbell, 1986). These mechanisms include monitors, constructs based on synchronous message passing and guarded commands, such as DR (Brinch Hansen, 1978) or CSP (Hoare, 1978), and path expressions (Campbell and Habermann, 1974).

In comparison, few synchronization mechanisms have been proposed for object-oriented systems, mainly because a few O–O systems support concurrency and shared objects. Emerald provides a 'monitor' construct allowing mutual exclusion to a set of data and associated methods. Trellis/Owl (Moss and Kohler, 1987) proposes a similar tool, based on locks and wait queues. In Hybrid (Nierstrasz, 1987), delay queues allow scheduling activities in the same way as condition variables in monitors. All these mechanisms are limited and do not allow canonical synchronization schemes to be expressed easily.

The 'mediator' (Grass and Campbell, 1986) is a powerful synchronization mechanism, based on guarded commands with statements allowing mutually exclusive service calls (the *exec* statement), or asynchronous calls (the *spawn* statement), and return with the *release* statement. These statements allow the activity scheduling within an object to be controlled. Guards express preconditions that can be programmed using the variables of the mediator, the service requested by the client (name of the service, value of the parameters and key identifying the client) and the status of the pending requests. This tool is very powerful, but is very difficult to implement efficiently. In addition, it is far too complex for most of the usual synchronization schemes. This analysis motivated us to define a mechanism with an expressive power between the locks and the mediators.

In Guide, synchronization is specified 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.

Synchronization is specified as a set of *activation conditions*, which are similar to Dijkstra's guarded commands: an activation condition is attached to a method, and must be satisfied before the execution of this method may start. This condition is expressed as a function of the internal state of the object and the parameters of the invocation. If no activation condition is attached to a method, then the execution of this method is unconstrained.

4.2. Basic synchronization model

In our language, the activation conditions are grouped in a **control clause** at the end of the class description. 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 and synchronization counters. Synchronization based on counters was introduced in another context by Robert and Verjus (1977).

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 run-time system. 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) = started(m) - completed(m) : number of
  current executions of method m
pending(m) = invoked(m) - started(m) : number of
  activities blocked on method m
```

In addition to this basic mechanism, activation conditions can be expressed by predefined synchronization policies (mutual exclusion and reader/writer schema).

The use of activation conditions is illustrated by the following example, which implements a buffer used for communication between activities (the type `Producer-Consumer` is supposed to have been introduced elsewhere):

```
CLASS FixedSizeBuffer IMPLEMENTS ProducerConsumer IS
  CONST size=100;
  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.
```

For each control statement, the compiler produces a prologue and an epilogue section which are attached to the corresponding method. This code is executed in mutual exclusion before entering and when leaving the method. The implementation of this synchronization scheme is described in Decouchant *et al.* (1991).

Since there is no limitation for subclassing, there is no rule for overloading an activation condition. The application programmer may choose either to inherit (and possibly complete a parent class definition), or to define a new activation condition.

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 cannot be directly expressed and must involve the use of additional methods.

4.3. Extended synchronization model

The above mechanism was used intensively for the construction of many classical communication and synchronization schemes. The experience drawn from these experiments showed that the use of the synchronization counters is sometimes cumbersome and thus prone to errors. For this reason, a number of shorthand forms and new mechanisms have been introduced. Some of these extensions are described now:

- the value of a counter associated to a method can be designated by the variable `my.counter_name`,
- the notation `current (op_1, op_2)` can be used instead of `current (op_1)+current (op_2)`,
- *macros* can be defined to factorize expressions within the control clause,
- *abstract counters* are counters associated to a set of methods.

To illustrate the expressive power of this extended synchronization model, we describe below the specification of a 'bounded buffer' communication scheme with various management policies. A `BoundedBuffer` class may be specified as follows:

```
CLASS BoundedBuffer IMPLEMENTS ProducerConsumer IS
  ...// same implementation as FixedSizeBuffer
  CONTROL
    NbItem = completed(Put) - completed(Get);
      // macro

    Put: NbItem <size AND current (Put)=0;
    Get: Nbitem >0 AND current (Get)=0;
  END BoundedBuffer.
```

The following class implements the 'Readers first' policy (priority is given to the get operations):

```
CLASS ReadersFirst SUBCLASS OF BoundedBuffer ...
  // Implementation is inherited
  CONTROL
    Put: INHERIT AND (pending(Get) = 0);
    // the control statement for Get is inherited
       by default;
  END ReadersFirst.
```

In the *Nopriority* policy, a Put request may only be executed after completion of all Puts and Gets prior to it. Conversely, a Get request may only be executed after completion of all Puts prior to it. When a Get is in progress, any incoming Gets are processed until a Put request arrives. We must therefore be able to memorize pending invocations of Puts during the execution of both primitives. This is achieved by using the value of invoked counter for each method.

```
CLASS NoPriority SUBCLASS OF BoundedBuffer ...
  // Implementation is inherited
  CONTROL
    Get: INHERIT AND completed(Put) =
      my.invoked(Put);
    Put: completed(Put,Get)=my.invoked(Put,Get);
END NoPriority.
```

The last example consists of adding a new method GetRear, which extracts a buffer item from the tail of the buffer. In addition, we want to allow GetRear and Get to run concurrently, as they do not operate on the same buffer item (note that this is allowed if and only if the current number of items in a buffer is greater than 1). If there is only one item in the buffer, GetRear is not allowed.

```
CLASS ExtendedBuffer SUBCLASS OF BoundedBuffer ...
  METHOD GetRear (OUT m: Element);
  BEGIN m := Buffer[last]; last := last-1;
    END GetRear;
  CONTROL
    NbItem = completed(Put) -completed(Get,
      GetRear)// macro
    // GetRear and Put use the same variable: last
    //GetRear and Get are not exclusive, they
      don't consume the same element
    GetRear : NbItem >1 AND current (Put, GetRear)
      =0;
    Get : NbItem >0 AND current (Get) =0;
    Put : NbItem <size AND current (Put, GetRear)
      =0;
END ExtendedBuffer.
```

Using abstract counters, the ExtendedBuffer activation conditions can be expressed as a specialization of the BoundedBuffer's activation conditions:

```
CLASS BoundedBuffer
CONTROL
  // abstract counter
  Produce = {Put};
  Consume = {Get};
  MdLast = {Put};
  MdFirst = {Get};

  // macro
  NbItem=completed(Produce)-
              completed(consume);

  Get: NbItem >0 AND current (MdFirst)=0;
  Put:NbItem<size AND current (MdLast)=0;
END BoundedBuffer.

CLASS Extended Buffer...
CONTROL
  // Produce and MdFirst are
  // inherited by default
  Consume = {Get, GetRear};
  MdLast = {Put, GetRear};
  // NbItem is inherited by default

  // Get and Put are inherited
  // by default
  GetRear : NbItem >1
      AND current (MdLast) =0;
END ExtendedBuffer.
```

As shown in these examples, activation conditions may be modified incrementally using the inheritance mechanism. Thus, the restriction applicable to a given method in a superclass can be supplemented by additional restrictions for the same method in subclasses.

5. EXCEPTIONS

Exception handling has been refined in the last decade and is now an integral part of most high-level languages. Exceptions provide a means to separate a 'normal' flow of control from an 'exceptional' one, where the semantics of 'normal' and 'exceptional' may be predefined in the language or specified by the programmer. The advantages are twofold: the textual separation of the exception handling code from the normal one greatly improves the structure and readability of the program, while the semantic separation ensures that the normal flow is stopped when an exception occurs, and may resume only after the proper exceptional handling code has been executed.

This section describes the main characteristics of exception handling in Guide. A complete description of the exception model is available in Lacourte (1991).

5.1. Exception declaration

Experience with exceptions in object-oriented languages is still limited. The integration of an exception mechanism in an object-oriented model should be consistent with the structuring principles of that model. Thus we think it is natural to associate the exception handling mechanism at the object invocation level. Exceptions in Guide are defined at the operation level, so that an object invocation can either terminate normally returning a result, or abort signalling an exception.

In our approach an exception can be viewed as an exceptional variation of the invocation, allowing the dialogue between caller and callee to be enriched. In guide, this dialogue is specified by the interface (i.e. the type signature) of the called object. Consequently the type model is extended to include exception declarations. This in turn affects the conformance rules checked by the compiler as exception are now part of the signature.

Guide exceptions are identified by symbolic names (system exceptions and hardware traps have reserved names). It is sometimes useful to attach complementary information to an exception. In Guide, this capability is achieved through the use of the output parameters in the invoked method This solution has the advantage of enabling the passing of parameters when signalling an exception, while keeping the simplicity of exception raising and the attachment of an exception to a type.

When an operation cannot fulfil its requirements, it raises an exception (using the keyword RAISE). Control is then transferred to a calling entity which provides an *exception handler* for this exception. The handler executes some code ('exceptional' flow of control) and then the 'normal' flow of control is resumed, according one of the following policies: the *resumption approach* resumes the execution at the level of the signalling entity; the *termination* approach resumes the execution at the level of the handling entity; the *propagation* approach signals a possibly different exception to a higher calling entity.

A termination model has been adopted in Guide: the interrupted method is stopped, the control is transferred to an exception handler which executes the code specified for this exception, and then execution is resumed at the level of the handler. An unexpected exception is transformed into a pre-defined system exception which is propagated to the invoking method.

5.2. Exception handling

Conceptually, handlers are associated with method invocations (and not with any block of statements such as in traditional approaches). A handler specifies a specific procedure for handling expected exceptions. A handler may also be associated with a class, in which case the specified handler is used for all methods of the class.

```
TYPE Page is
  METHOD Getchar : Char;
    SIGNAL end_of_line, end_of_page;
  END Getchar;
END Page.

CLASS Editor IMPLEMENTS Editor IS
  METHOD PrintPage; // of class Editor
  BEGIN
    WHILE TRUE DO output.WriteChar(currentPage.
      GetChar);
    EXCEPT
      end_of_line FROM Page: REPLACE '\n';
        //gives a newline character
      end_of_page FROM Page: RETURN;
    END PrintPage;
END Editor.
```

In this example, the handlers defined in the EXCEPT clause are syntactically associated with the two method invocations of the WHILE statement. This usually allows exception handling to be factorized for a number of method calls. However, when it is necessary, the keyword FROM allows exceptions raised by invocations on various object types to be distinguished (e.g. here the handler is associated only with exceptions raised by the invocation of the GetChar method of the type Page).

The keyword REPLACE allows a handler to substitute a result to a failed call. When the call currentPage.GetChar raises an end_of_line exception, the handler supplies a replacement value (replace is discussed later) and PrintPage continues with a call to output.WriteChar to print it. The conformity relationship between the replacement value and the result expected from the operation is checked at compile time.

In some cases, a handler may decide to reexecute the failed invocation. This is achieved using the RETRY keyword. This capability allows general fault recovery policies to be specified. To prevent the risk of infinite loops due to recursive calls to the same handler, the system ensures that a handler cannot be called again while it has not terminated.

In the following example, the handler associated with the class Editor tries to recover from a failure of the ObjectManager (called in method CreateNewPage).

```
CLASS Editor IMPLEMENTS Editor IS
  METHOD Garbage;
  page: REF shadowPage;
  BEGIN
    page := shadowPages.First;
    WHILE page DO
      IF page.isDirty
      THEN shadowPageManager.Free(page); page :=
        shadowPages.Delete;
      ELSE page := shadowPages.Next;
    END;
  END Garbage;
  EXCEPT
    noObjectLeft FROM ShadowPageManager.Get:
      SELF.Garbage; RETRY;
  END;
END Editor.
```

5.3. Restoration

Our exception handling mechanism allows application programmers to implement alternate or retry policies as needed. However, it does not ensure that the called object is in a consistent state after raising an exception. Therefore the RESTORE keyword allows a restoration block to be defined. This block is executed only if the method exits abnormally (raising an exception). The block is executed just after the raising of the exception and prior to the execution of the handler. Recursively, if the handler propagates an exception, then a restoration block of the calling object is executed before the search for a new handler. A restoration block may be associated with a given method or with the entire class.

In the following example, a restoration block associated with the class Editor allows the dirty Pages of the Editor to be saved whenever it exits abnormally.

```
CLASS Editor IMPLEMENTS Editor IS
  ...
  RESTORE
  page: REF ShadowPage; shadowDocument;
    REF Document;
  BEGIN
    page := shadowPages.First;
    WHILE page DO
      IF page.isDirty
      THEN shadowDocument.pages.Append(page);
      ELSE shadowPageManager.Free(page);
```

```
        page := shadowPages.Delete;
      END;
      IF shadowDocument.pages.nbItem > 0 THEN
        output.WriteString(''abnormal exit\n'');
        output.WriteString(''modifications saved
          in'' + <name>);
          ...
      END;
    END;
  END Editor.
```

6. EVALUATION

The Guide prototype—including the Guide language and its run-time system—has been intensively used both inside and outside of the Comandos consortium for experimentation. Altogether the pilot applications built on top of Guide represent about 150 000 lines of Guide source code, 60% of which have been developed outside of Grenoble. These applications include a number of simple demonstrators aiming at illustrating specific features of the Comandos model, as well as large scale real world distributed applications in the field of office and business systems. They include:

- Services, such as a distributed directory service, compliant with X-500, and a distributed monitoring facility, primarily targeted at the observation and debugging of a distributed environment.
- A distributed mail system, and a distributed diary.
- A multi-user distributed spreadsheet.
- A cooperative document editor, which allows the cooperation of several authors for the production of a common document in a distributed environment.
- A workflow application dealing with the intelligent circulation of documents and folders within an enterprise (see chapter 8 of Cahill *et al.*, 1993).

This section intends to draw some lessons from these experiments. In a first step we briefly analyse some statistics regarding the use of objects, types and classes. Then we summarize the main conclusions from a qualitative point of view.

Table 1 gives some preliminary figures about the use of types and classes. For each category of applications, Table 1 gives the number of types, the number of classes and the number of lines of source code.

Object granularity has always been a big debate as it strongly influences the overall architecture of the run-time system. From our experience, it appears that objects are usually small, as demonstrated by the figures below:

- *Classes*: 63% of them are smaller than 10 Kbytes (and 86% smaller than 15 Kbytes); this is mainly due to the fact that a Guide class does not contain inherited code or Unix library code which are linked dynamically.
- *Objects*: 89% of them have a size less than 512 bytes and only 5% have a size greater than 2 Kbytes.

TABLE 1.

	No. of types	*No. of Classes*	*Lines of code*
Services	177	171	52 300
Bull-IMAG applications (mailer, diary and spreadsheet)	78	79	27 100
Workflow and cooperative editor	163	120	58 500
Test programs	251	269	18 000

In the rest of this section, we point out the major benefits expected from the use of the Guide model and language, based on the current experience.

- *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 the same interface allows easy and transparent implementation modifications, while provision of multiple interfaces for the same class, via type restriction, is a very natural way of filtering access to a shared data structure.
- *Execution model.* The execution model provides a higher abstraction level than the simple client–server model which is supported in classical approaches. On the one hand, this facilitates the design of a distributed application; on the other hand this leads to more structured and modular programming. In addition, the execution model allows an easy modelling of sophisticated communication and synchronisation policies. The approach considered in Guide, which consists in defining activation conditions and exception handling at the object invocation level, is consistent with the object paradigm and greatly assists the programmer in structuring and debugging her application. In addition, the availability of predefined synchronization policies (e.g. mutual exclusion, reader/writer) allowed most of the usual cases to be covered without any additional code.
- *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.
- *Distribution 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. All the applications were 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.

- *Multiple granularities of objects.* The Comandos technology supports multiple granularities of objects. The use of references allows the construction of complex objects as composed of smaller objects. Furthermore a given object may be shared between different compound objects. In addition, any of the components of a complex object can be distributed over the network. This is especially suitable for supporting distributed hyper-text (or hyper-media) applications.

To summarize, shared, transparently distributed objects, with high level language support, were considered an extremely useful tool by the programmers of distributed applications, in contrast to explicit messages. In addition to the benefits specific to the language (strong typing, conformity, multiple implementations of a type, etc.), the main advantages mentioned were: the higher degree of abstraction for the expression of distribution (an application developed on a single node could be ported to a network without change); the ability to build large structures with embedded object references and the ability to share substructures; the separate expression of synchronization constraints for shared objects; and the implicit management of persistence.

The users also had some critical remarks. Programming with persistent objects needs a change in programming practice; while transparent distribution is convenient, it also has less desirable aspects (debugging is more complex; some applications still need explicit control on object location); the purely synchronous invocation model has its limitations (dealing with asynchrony is possible but leads to awkward constructs).

7. CONCLUSION

A major advantage of the Comandos technology is the combination of the well-known advantages of the object technology with advanced features for the expression and control of distributed and cooperative computations.

The provision of *programming language support* that integrates these features was a key aspect of the Comandos project. Two different approaches to the provision of programming language support were considered in the framework of the project: the use of existing languages and the provision of a new language environment called Guide.

This paper has presented the basic features of the Guide object-oriented language, as well as preliminary lessons gained from its use for programming distributed cooperative applications. Two prototypes of the language and its run-time environment are currently available: the first prototype (named Guide-1) runs on top of Unix; a second prototype (named Guide-2) is built as a server on top of the Mach 3.0 micro-kernel and interworks with the OSF-1 server. Both prototypes provide full coexistence with a Unix environment, so that existing applications can be reused.

The compiler for the Guide language produces C code which is in turn compiled by a standard C compiler. This approach has two main advantages: on the one hand it allows C code to be easily integrated within class definition; on the other hand it allows the Guide compiler to be easily ported. For each type, the compiler produces a type descriptor which is used during the compilation of other types and classes.

For each class, the compiler produces the corresponding C code and a class descriptor which is used in the compilation of further classes. Descriptors are stored in the Unix file system. The output of the C compiler is post-processed to produce a Guide binary, which is stored in the Guide storage system.

To assist the development process, a set of development tools have been designed and implemented. These tools are fully integrated into a coherent development environment through the use of control integration mechanisms using Guide abstractions. The cooperation between the tools within this integrated environment allows a rapid development and debugging of distributed applications. The tools currently available are: a desktop that provides a uniform access to the development facilities; a syntactic editor for the Guide language, based on Emacs; a type and class browser; the Guide compiler; an error browser; and an observer and a distributed debugger.

In the near future it is intended to work in the following directions:

- Revision of the type model to be compliant with the Interface Definition Language (IDL) specified in OMG's CORBA. In addition it is intended to use the IDL type model as an intermediate representation to allow interworking between Guide programs and C++ programs.
- Revision of the execution model, based on our experience, and to better integrate cooperative applications requirements.
- Consolidation of the development environment.

REFERENCES

Accetta, M., Baron, R., Bolosky, W. *et al.* (1986) Mach: a new kernel foundation for Unix development, In *Proc. Summer Usenix Conf.*, pp. 93–112, Atlanta, GA.

Albano, A., Cardelli, L., and Orsini, R. (1985) Galileo: a strongly typed, interactive conceptual language. *ACM Trans. Database Syst.*, **10,** 00–00.

Balter, R., Bernadat, J., Decouchant, D. *et al.* (1991) Architecture and implementation of Guide, an object-oriented distributed system. *Comput. Syst.*, **4,** 31–67.

Bancilhon, F., Briggs, T., Khoshafian, S. and Valduriez, P. (1987) FAD: a powerful and simple database language. In *Proc. 13th Very Large Data Base Conf.*, Brighton.

Black, A. P. (1985) Supporting distributed applications: experience with Eden, *10th ACM Symp. Operating Systems Principles. SIGOPS Operating Syst. Rev.*, **19,** 181–193.

Black, A. P., Hutchinson, N., Jul, E., Levy, H. and Carter, L. (1986) Distribution and abstract types in Emerald. *IEEE Trans. Software Eng.*, **SE-12,** 00–00.

Brinch Hansen, P. (1978) Distributed processes: a concurrent programming concept, *Commun. ACM*, **21,** 934–941.

Cahill, V., Balter, R., Harris, N. and Rousset de Pina, X. (eds) (1993) *The Comandos Distributed Application Platform.* Springer-Verlag, Berlin.

Campbell, R. H. and Habermann, A. N. (1974) *The Specification of Process Synchronisation by Path Expressions, Operating Systems, Lecture Notes in Computer Science.* pp. 89–102, Springer-Verlag, Berlin.

Cardelli, L. and Wegner, P. (1985) On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, **17,** 471–522.

Dasgupta, P. *et al.* (1990) The design and implementation of the Clouds distributed operating system. *Comput. Syst.*, **3,** 11–46.

Decouchant, D., Le Dot, P., Riveill, M. *et al.* (1991) A synchronisation mechanism for typed objects in a distributed system. In *Proc. 11th Int. Conf. on Distributed Systems (ICDCS)*, pp. 152–159, Arlington, TX.

Freyssinet, A., Krakowiak, S. and Lacourte, S. (1991) A generic object-oriented virtual machine. In *Proc. 2nd Int. Workshop on Object Orientation in Operating Systems*, Palo Alto, CA.

Goldberg, A. (1985) *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, MA.

Grass, J. E. and Campbell, R. H. (1988) Mediators: a synchronisation mechanism. In *Proc. 6th Int. Conf. Distributed Systems (ICDCS).* pp. 453–457, Cambridge.

Hoare, C. A. R. (1978) Communicating sequential processes. *Commun. ACM*, **21,** 666–677.

Lacourt, S. (1991) Exceptions in Guide, an object-oriented language for distributed applications. In *Proc. Eur. Conf. on Object-Oriented Programming (ECOOP)*, Genève.

Lalonde, W. (1991) Subclassing ≠ subtyping ≠ Is-a. *J. Object Oriented Program.*, **3,** 11–22.

Liskov, B. (1985) The Argus language and system. In *Distributed Systems: Methods and Tools for Specification*, Paul, M. and Siegert, H. J. (eds), pp. 343–430. Springer-Verlag, Berlin.

Meyer, B. (1988) *Eiffel: the language.* Prentice-Hall, Englewood Cliffs, NJ.

Moss, J. and Kohler, ?. (1987) Concurrency features for the Trellis/Owl language. In *Proc. Eur. Conf. on Object-Oriented Programming (ECOOP)*, Location?

Mullender, S., Van Rossum, G., Tanenbaum, A. S., van Renesse, R. and van Staveren, H. (1990) Amoeba—a Distributed System for the 1990s. *IEEE Comp.*, **May,** 000–000.

Nierstrasz, O. M. (1987) Active objects in hybrid. In *Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 243–253, Location?

Robert, P. and Verjus, J. P. (1977) Toward autonomous descriptions of synchronisation modules. *Proc. IFIP Congr.*, Gilchrist, B. (ed.), pp. 981–986, North-Holland, Amsterdam.

Schaffert, C., Cooper, T. and Carrie, W. (1985) *Trellis Object-Based Environment, Language Reference Manual, DE-TR-372.*

Shapiro, M., Gourhant, Y., Habert, S. *et al.* (1989) SOS: an object-oriented operating system—assessment and perspective. *Comput. Syst.*, **2,** 287–338.