The Object and Event Oriented Language TOOL

Sergio Eduardo Rodrigues de Carvalho

Departamento de Informática, Pontifícia Universidade Católica RJ

Rua Marques de São Vicente 225, Gávea, Rio de Janeiro, 22453

Telephone: (55)(21)274-4449 Fax: (55)(21) 511-5645

e-mail: sergio@inf.puc-rio.br

ABSTRACT

TOOL is a programming system designed to simplify the task of building application programs running on graphical user interface platforms. The current version runs above Microsoft Windows, version 3.1. This report describes the programming language component of the system. This is an object and event oriented language, designed to be easily understood by conventional language programmers. In this language both the synchronous and the asynchronous behavior of objects are declared in classes.

KEYWORDS AND PHRASES

programming language, object orientation, event driven, message passing, class, object

ACKNOWLEDGEMENTS

The TOOL Programming System is the result of a joint research and development project conducted by SPA (Sistemas, Planejamento e Análise), a software house in Rio, and a group from the Departamento de Informática of the Pontificia Universidade Católica, also in Rio. The system contains a programming language and its compiler, a linker, an abstract machine and run-time support, a library of classes, an environment for the friendly construction of TOOL programs, and an interactive application constructor. This report must be understood as another result from this same team effort, although the author takes sole responsibility for its contents. Special thanks are due to Otavio P. Coelho, Nelson Gorini, and Werther J. Vervloet, early day team members. Thanks are also extended to the younger generation, Mario S. Cardoso, Eduardo Galucio and Toacy C. Oliveira, and to several PUC students and colleagues who contributed to the production of this report.

INTRODUCTION

Software development platforms for the construction of applications driven by graphical interfaces are becoming increasingly popular (Windows for DOS [1] and MOTIF for Unix [2] are examples). Application users benefit considerably from this technique: for example, the fact that application interfaces tend to be similar speeds up the learning process, allowing users more productivity in less time.

What is nice for users, however, is not necessarily so for programmers. The task of developing a Windows application in C, for example, has been recognized as at least non-trivial. For this reason new programming systems for Windows-like platforms are being developed and marketed today. All support graphical user interfaces and most advertise object orientation, a definitive trend for the 90's. We can roughly classify most of these new systems in two categories, described in some detail below.

In the first category we place those systems developed as extensions to well-known conventional programming languages, notably C and Pascal. Most conventional or procedural languages, however, have not been originally designed to accommodate asynchronism, or event-driven programming, essential to graphical user interfaces. As a result, when using one such language to construct an application for a modern software platform, one usually has to learn a new, large and unstructured set of functions, in order to code message passing. Moreover, one frequently has to cope with inelegant syntactic and semantic language extensions, which often violate original design principles.

In the second category we place those systems presenting object orientation in such a way that completely new programming language features must be used in program development, as is the case of Smalltalk [3]. It turns out that these new features are unfamiliar to most programmers today. As a result, procedural language programmers trying to embrace object orientation often feel alienated, thus failing to benefit from the obvious advantages offered by graphical user interface platforms.

Neither approach seems adequate to us. What seems to be needed are new languages, certainly built for the resources and needs of today (thus including object and event orientation, for example), and yet positively utilizing the extensive body of programming knowledge existing today (thus welcoming all conventional language programmers). In this paper we present one such language, TOOL [4]. TOOL accommodates, through object orientation, both the message passing and the procedural paradigms. In *classes*, the main language construct, both the synchronous (procedural) and the asynchronous (event-driven) behavior of objects may be specified.

TOOL is actually an object and event oriented programming system, consisting basically of a programming language, an optimizing compiler, a linker, a library system, an abstract machine to allow portability through different environments, and a friendly program construction interface. The current version of the system is implemented as a software layer above Microsoft Windows, version 3.1.

Our main goal, when designing TOOL, was to simplify the task of building application programs using the extensive facilities provided by Windows and similar environments. To

accomplish this, considerable effort has gone into the design of programming language constructs that, besides implementing modern software features such as object orientation, message passing, and graphical user interfaces, seem also familiar to conventional language programmers. Thus, TOOL classes are presented as type extensions; message passing, as a special case of the familiar procedure call; polymorphism, simply as an ability objects have of changing classes at execution time.

Next we present the main features of the TOOL programming language. The TOOL programming units are *classes*, for modeling objects and other classes; *methods*, to implement synchronous object behavior; message *handlers*, to implement asynchronous behavior. Methods and handlers are located within classes.

CLASSES

Classes are the meeting grounds for all properties associated with the objects they model. Classes can be *basic, conventional,* or *extended*.

A collection of basic classes is supplied with the system. They include implementations for INTEGER, REAL, CHARACTER, BOOLEAN, and STRING. Conventional classes are user-declared classes and are not significantly different from classes found in conventional object oriented languages. These are discussed next. Extended classes model message passing and receiving objects, are also user-declared, and are presented at the end of this paper.

Conventional classes may contain an inheritance clause, an object representation section, a class representation section, class constants and methods.

Inheritance in TOOL is simple, establishing both a supertype-subtype relationship and also visibility regions. Thus, if B is a subclass of A, every B object is also an A object and may appear wherever an A object is expected; and every A attribute is visible in B, perhaps with qualification, if a same name attribute is declared in B. A class and its descendants constitute a *family*. To illustrate, we can imagine the need for a model for employees in an environment where a model for persons already exists. If we decide that in the application to be developed every employee is a person, then we can declare a new class Employee reutilizing the Person class already available:

```
CLASS Employee FROM Person;
        ...
END CLASS
```

Inheritance establishes the first of four possible relationships that may exist between two TOOL objects: the *subtype relationship*. In the example above, if we declare an Employee 'e', then we say that 'e' *is-a* Person.

In the *object representation section* the data structure for class objects is declared. This is basically a record consisting of object and structure declarations. Unless declared as public, representation components are only visible inside their family. For every component an initial value is given: either explicitly, or transitively obtained by default, since every basic class in TOOL has an initial value for its objects. The ordered collection of such initial component

values is a constant value of the class, called INITIAL. To exemplify, consider a model for points in a plane, represented by a pair of coordinates 'x' and 'y':

```
CLASS Point;
        REPRESENTATION
                REAL x, y;
        END REPRESENTATION
        ...
END CLASS
```

Every point modeled by the class above has two components, 'x' and 'y', both with value 0.0, which is the initial value for reals in TOOL. Thus the INITIAL value for every declared point is the pair (0.0, 0.0). However, each component in a representation section may be explicitly initialized, and thus the INITIAL value may be set according to the application's needs.

The object representation section of a class declaration introduces the second relationship that may exist between two TOOL objects: the *composition relationship*. In the example above we say that a declared point 'p' *has-a* component 'x' and *has-a* component 'y'.

In the *class representation section* the data structure for the class itself is declared. This is a statically allocated storage pool, accessible only from operations within the family. Class representation components, which are objects and structures, can be used as shared storage among objects of the class. To illustrate, consider the need to count the number of times a given operation, say Sum, is invoked during an application using points. The following section could be defined in class Point:

```
CLASS Point;
        ...
        SHARED
                INTEGER sum_count := 0;
        END SHARED
        ...
END CLASS
```

Special values for class objects may be declared in the *class constants* section. A constant object declaration associates a name with an immutable value. Such values may be assigned to and compared with variable objects of the class, thus saving time by eliminating the need for more expensive method applications. To consider again our running example, we may need the special points zero and one:

```
CLASS Point;
        ...
        CONSTANT
                zero = (x = 0.0, y = 0.0);
                one = (x = 1.0, y = 1.0);
        END CONSTANT
        ...
END CLASS
```

'zero' and 'one' are constant objects of the class Point, and the values of their components cannot be changed during execution.

Methods are active program units used in the implementation of synchronous operations on objects. They correspond to the notions of procedures and functions in conventional languages, and will be separately discussed below.

We summarize the declaration of a conventional class:

```
CLASS  name
        -- inheritance clause
        -- object declaration section
        -- shared declaration section
        -- constants section
        -- method declarations
END CLASS
```

OBJECTS

Objects are instances of basic, conventional or extended classes. Objects can be declared in the object, class and constant representation sections of a class declaration, as seen above; they can also be declared as operation parameters, as method results, and as locals to operations. Objects play in TOOL the same role variables play in languages like C and Pascal: they are execution time entities, modeled by their declaration classes, as variables are modeled by their declaration types.

At declaration, which is always required, an object is associated with a *base class*, and is modeled by this class. At declaration, a *discipline* is also attached to objects. This

discipline affects both the lifetime and the class flexibility of objects. The possible disciplines are *automatic*, *dynamic* and *polymorphic*.

The *automatic* (default) discipline is that of languages like Pascal: for local, parameter or result objects the space for the object's data structure is allocated on the execution stack, and the object's lifetime is that of the enclosed operation. For objects declared as components in a representation section, the lifetime is that of the parent object.

Objects with the *dynamic* discipline have their lifetimes controlled by the programmer, through the application of built-in methods for creation and disposal. In addition, a set of special methods to safely control dynamic objects exists in the language, avoiding dangling references and simplifying garbage collection. In all other respects dynamic objects behave exactly like automatic objects.

The lifetime of *polymorphic* objects is also programmer controlled. Unlike automatic and dynamic objects, however, they have the ability of changing classes during execution. The set of possible new classes for such objects is the object's family. Polymorphism results from the application of a special method to a polymorphic object. The method's argument must be an already created dynamic or polymorphic object of the receiving object's family. After the method's application, the receiving object shares the representation of the argument, and is of the same class as the argument.

Basic class objects are always automatic. Extended class objects cannot be automatic. Any discipline is suitable for conventional class objects.

STRUCTURES

Object structures can be constructed with the *array*, *dynamic array* and *union* mechanisms. Structures are merely compositions of objects; they are not classes. Structures may appear in the object and class representation sections of a class declaration. They may also be locally declared in operations. They may not, however, appear as parameters or results in operation interfaces; to accomplish this, one would have to encapsulate a structure within a new class, probably having no other section in its declaration, and pass or receive back an object of this class.

Arrays are homogeneous fixed-size compositions. Access to array components is done by indexing the array's name with integer expressions. Array components can be of any class and have any suitable discipline. *Dynamic arrays* are created during execution, where special methods are able to set and then get dimension limits. *Unions* are probably heterogeneous compositions where all elements share the same memory positions. Union elements, which are basic class objects or one dimensional arrays thereof, are accessed with dot notation.

METHODS

Methods have a signature, local declarations and statements. The signature of a method, besides the method's name, optionally contains the specification of parameters, of a result object, and of a method attribute. Attributes exist for in-line code expansion, for private methods, and for methods which cannot be fully declared in a class, but only in its subclasses.

Parameters and *arguments* correspond according to position, family, mode and discipline. Parameters can be objects or merely argument identifiers. Object parameters

correspond either through the assignment of a value (input, output or both), or through the passing of a reference (input or output). In both cases a subclass object may correspond to a superclass object. Object parameters can have any suitable discipline. Dynamic and polymorphic parameters are automatically created by the system. Argument identifiers are merely local aliases for the corresponding arguments in method applications.

The method's *result* is locally expressed with the declaration of an object, which can receive values during the method's execution. The last such value is the result of the method's application. Any suitable discipline can be attached to the result object.

We exemplify TOOL methods by giving some behavior to points:

```
METHOD GetAbcissa RETURNS REAL r;
BEGIN
        r :=  SELF.x;
END METHOD  -- GetAbcissa.

METHOD  Sum ( IN Point z ) RETURNS Point c;
        -- invoked by a <- Sum(b)
BEGIN
        c.x := SELF.x + z.x;
        c.y := SELF.y + z.y;
        sum_count := sum_count + 1;
END METHOD  -- Sum.
```

Methods can be redeclared in subclasses. In this case the same signature is required. Methods are applied to *receiving objects*, specially designated in method body statements by

SELF. Receiving object components are accessed with dot notation from this special designator. The components of other local objects of the same class are also directly accessed with dot notation from their names. Components of objects of other classes that have the PUBLIC attribute in the declarations of those classes, are also accessed with dot notation. Private components of objects of other classes are not visible inside the method.

A set of built-in methods is available in any class: for object creation and disposal, and for polymorphic transformation, for example. Certain built-in methods can only be applied to objects satisfying a certain declared discipline. Examples include creation/disposal methods, applicable to dynamic or polymorphic objects only, and the polymorphic transformation method.

STATEMENTS

A small and conventional looking set of statements is available in TOOL: assignment, selection, repetition, operation application, and termination of loops and operations.

In an *assignment* a left hand side object receives a value that is either obtained from an expression or is the value of an object (variable or constant). In all cases the right hand side value must be of the left hand side declaration class, or descendant thereof. Only common declaration class fields are updated.

Selection is expressed by conventional **if** ... {**elsif** ...} [**else** ...] , and **case** ... {**when** ...} [**else** ...] statements.

A single **loop** statement is used to specify *repetition*. The execution of this statement can be controlled by conditional **exit** and **repeat** statements placed anywhere in the loop's body. The exit statement terminates the loop's execution, transferring control to the statement immediately following the loop. The repeat statement terminates a loop iteration, transferring control to the beginning of the loop's body.

Application statements force operations (methods and message handlers) on receiving objects. Object returning methods, when applied, produce object values to be used in expressions. Handlers, to be examined below, do not produce values. A conditional **return/quit** statement may be used in method/handler bodies to abruptly terminate their execution.

PARTIAL CONCLUSIONS

The TOOL features presented so far, namely basic and conventional classes, objects, structures and statements, are to a certain extent typical of object oriented languages. We could mention as exceptions the declaration, in classes, of constant objects, and polymorphism as a discipline users may decide to attach to objects at declaration. However, in order to create a modern language for graphical user interface applications, where users can asynchronously interfere with the execution, new features are needed. A central point in these features is the handling of asynchronous messages. This is the concern of the next section.

EXTENDED CLASSES

Extended classes model objects capable of making and responding to asynchronous calls. This modeling is done at a very high level, taking full advantage of the class mechanism: TOOL programmers are not concerned with message queues, for example.

The basic message passing organization in TOOL has to do with the declaration of a special kind of operation, a *message handler*, in an extended class. The declaration of a handler for a message M in a class modeling an extended object 'p' states that 'p' is able to receive the message M; this is part of the declared behavior for 'p'. The reception is done with the creation of a local environment for the handler and with the execution of the handler's statements. This execution is asynchronous, moreover: the operation (method or handler) that was executing when the message M was applied to 'p' continues after the message is sent, never expecting any results from the execution of the corresponding handler.

To illustrate, consider a class P containing a handler for a parameterless message M:

```
XCLASS P;
        ...
        HANDLER FOR M;
                -- local declarations, the handler's environment.
        BEGIN
                -- statements, the handler's body.
        END HANDLER
        ...
END XCLASS
```

If we also suppose the existence of an object 'p' modeled by the extended class P, and the message application statement

p <<- M;

then the handler for M declared in P would eventually be executed on the receiving object 'p', thus completing the message sending/receiving cycle.

The cycle explained above, which could be called the named message cycle, involves the sending of messages to known objects, or objects whose names are known in the message sending context. An important feature of the full message passing mechanism in TOOL could, in contrast, be called the generic message cycle, this one involving a sending object and its owner. In order to explain the correspondence between an object sending a message and its owner, the *ownership relation* must be presented. We do this with an example.

```
XCLASS Interface;
        REPRESENTATION
                Window      w;
                PushButton  ok, cancel;
        END REPRESENTATION
        METHOD  R;
        BEGIN
                w <- CREATE;
                ok <- CREATE;
                cancel <- CREATE;
                ...
        END METHOD
                ...
END XCLASS
```

In the example above, when the method R is applied to an object of the class Interface, the window 'w' and the two buttons 'ok' and 'cancel' are created. This states that this interface object is the owner of 'w', 'ok' and 'cancel'. This is important, since the owner of an object that is able to send a generic message is the one responsible for handling the message sent; in other words, its class, or any superclass thereof, must contain the declaration of a message handler that is executed when that message is received.

Again in the example above, when the user presses a button, say 'ok', the PushButton class message 'Clicked' is sent. In the example, it is the responsibility of the class Interface, or of one of its superclasses, to contain a handler for this message. Actually, since two buttons with obviously different purposes have been declared in the Interface class, we would need two handlers for the same 'Clicked' message, one for each button. With this new behavior attached, the extended class Interface would look like this:

```
XCLASS Interface;

        REPRESENTATION
                Window      w;
                PushButton  ok, cancel;
        END REPRESENTATION

        METHOD  R;
        BEGIN
                w <- CREATE;
                ok <- CREATE;
                cancel <- CREATE;
...     END METHOD
```

```
        HANDLER FROM ok FOR Clicked;
            -- locals and actions
        END HANDLER

        HANDLER FROM cancel FOR Clicked;
            -- locals and actions
        END HANDLER
        ...
END XCLASS
```

We take further advantage of the example above to introduce the last relationship that may exist between TOOL objects: the *parenthood relationship*. This occurs between visual objects, or objects appearing in application interfaces. In the example, suppose the existence, in the method R, of the statement

```
ok <- PushButtonInitialize (...., w);
```

which defines argument values for the descriptor of the 'ok' button. The last such argument is the window 'w'. This establishes 'w' as the parent of 'ok', in the sense that both the visibility and the life cycle of the button depend on the window: if the window is iconized, the button ceases to be visible; if the window is destroyed, the button is destroyed as well.

As another message passing example, consider the need for buttons that know their positions; buttons used, for example, to represent chess or checker board cells. The extended class below simulates this situation.

```
XCLASS GameButton;
        REPRESENTATION
                SHORT  pos;  -- of button in board.
                PushButton   b;
        END REPRESENTATION

        MESSAGE   -- sent to owners
                GBClicked ( SHORT );
        END MESSAGE

        HANDLER FROM b FOR Clicked;
                OWNER <<- GBClicked ( SELF.pos );
                b <-  SetText ( "played" );
        END HANDLER

        METHOD GBInitialize (...);
                b <- CREATE:
                ...
        END METHOD
END XCLASS
```

In this model, the following behavior is specified:

- When a gamebutton is initialized in the application, its pushbutton component 'b' is created, as seen in the method GBInitialize. This establishes ownership, so it is in the GameButton class that messages sent from the pushbutton 'b' are to be handled.
- When the user of the application presses a pushbutton, a message Clicked, declared in the built-in PushButton class, is sent to the gamebutton that owns it.

- When this message is received, the handler for this message is activated, resulting in turn in the sending of the user-declared message GBClicked, with the button's position, to the owner of the gamebutton, and in changing the button's text to "played".

In this last example, a message section was added to the GameButton class. In this section all messages sent from objects of this class to their owners are declared.

It is now possible to see the generic message cycle in TOOL. In extended classes, one can declare not only the behavior of objects when a message is received, but also the sending of messages to generic owners, other objects established, at execution time, when some application is being run. We note that the message sending behavior of extended objects is thus independent of the application where they are placed; they merely send a message to the current owner, responsible for its handling. This truly favors encapsulation and class reutilization; gamebuttons, for example, could be reused wherever the need for their behavior is felt, provided that in their new context, the classes modeling their owners possess handlers for the messages they send.

We now summarize the semantics of extended classes. To encompass message passing, extended classes contain, besides the sections existing in conventional classes, a *message* section and the declaration of *message handlers*. The message section contains the signatures of all messages sent from extended class objects to their owners, or the objects that created them. Message signatures have a name and an optional set of parameters.

Handler declarations contain a signature, local declarations and statements. Handler signatures contain a message identification clause and an optional sender identification clause. The message identification, always required, identifies a message that objects of the extended class being declared can respond to, by creating the corresponding handler environment and by executing its statements.

The sender identification, when present, indicates a set of owned objects that are possible senders of the message to be handled, and which will cause this same message handling. All objects in this clause must be modeled by the same extended class. Handlers without a sender identification clause treat messages originating from non-owned objects.

Handler applications are syntactically similar to method applications. The semantics differ, however: handlers do not return to the calling environment, they just eventually quit executing; handler parameters are input only, and handlers do not return values.

In order to generalize the passing of a message from an object to its owner, a generic OWNER identifier exists in the language. In this way the behavior of objects can be specified independently of the context in which they are placed, truly favoring encapsulation.

A PROGRAM

A TOOL program is a sequence of one or more conventional and/or extended classes. Some extended class in the sequence must be identified as the main class (this is done through the program construction interface, a Windows application supporting the construction, compilation, linking and execution of TOOL programs). The main class must contain a

method called Main. Execution starts with the application of Main to an extended object of the main class, automatically created by the system.

Actually one or more extended classes in the sequence defining a TOOL program may have methods called Main. Due to this a TOOL program may have a set of entry points. Once one of these classes is identified as the main class, however, the program construction is set and cannot change.

TOOL AND OTHER LANGUAGES

In this section we compare some important TOOL features with their counterparts in other object oriented languages. We recall that this system was designed to simplify the task of programming with objects in graphical interface and message passing environments, and that the familiarity of language constructs was considered to be of primary importance in reaching this goal.

We begin with the class concept, discussing conventional classes, which in TOOL model objects responding only to procedure calls (objects also responding to messages will be treated separately). Conventional classes in TOOL, as the classes in EIFFEL [5], are merely object models, and do not behave as objects in any way, as is the case in Smalltalk and even in C++ [6]. In particular, we feel confortable in being able to apply to an already declared object a class method called CREATE, which allocates, at execution time, space for the object's descriptor in the heap. We do not feel the need to introduce meta classes and object constructors in our system, considering these devices unnecessary complications in the

understanding of object orientation. With this simplification we can stress familiarity by stating that TOOL classes model objects in the same way that Pascal types model variables.

Still with respect to conventional classes, we decided to adopt simple inheritance. This simplifies the language, which does not need, for example, renaming facilities to disambiguate attribute accesses. Furthermore, this avoids the cancellation of superclass functionalities, corresponding to unwanted superclass operators, by redeclaring those unwanted operators with empty bodies. This latter argument was considered of primary importance, since in our view programmers could quickly lose confidence in inheritance, one of the new object orientation mechanisms, if only partial behavior was to be actually inherited. We note in passing, however, that this kind of cancellation can still occur in TOOL; we only feel that its need is much reduced with simple inheritance.

The object representation section in a TOOL class has basically the same semantics as the one modeling instance variables in Smalltalk or type attributes in EIFFEL: it is simply a record describing, to the compiler, the space organization needed for each object of that class, at execution time. The class constants section, on the other hand, has no equivalent in those languages: in this section we can define objects having immutable values in their components. We can examplify by considering the constant objects 'zero' and 'one', which could be declared in a class Point. This is especially useful in assignments and comparisons, avoiding more expensive method applications.

Other distinguishing TOOL features are the class binding and life cycle disciplines that are related to objects at declaration. We allow programmers to choose between the static and

dynamic class bindings for their conventional objects. In this way one can use conventional class objects as if they were Pascal variables, allocating them on the execution stack (the *automatic* discipline), or in the heap (the *dynamic* discipline). In either case the class binding is static, and as a consequence the compiler can generate efficient, tight code; in either case, moreover, these are disciplines found in procedural programming languages, and therefore familiar.

The *polymorphic* discipline, the default discipline in most object oriented languages, is in TOOL chosen by the programmer. In this way the high price of consulting, at execution time, a table containing all methods possibly applicable to a polymorphic object, in order to choose, according to its current class, the one actually applicable, is paid only when due.

There is another important observation that can be made with respect to polymorphism in TOOL. By associating this feature with objects, and not, as is more usual, with methods, one gains familiarity: the idea of having an object changing classes at execution time seems to be more acceptable to programmers trying to enter the object orientation world than that of methods "changing form" at execution time, since even prior to execution all methods have already been defined and reside in their classes, thus having their "forms" well set.

EXPERIENCE

The development of the TOOL Programming System has been under way for almost five years. As soon as code was being generated experience in the use of the language began. The first task was the construction of a library of classes for the system. This library

should provide users with the entire functionality of the Windows operating system, by arranging the hundreds of functions available in Windows, for example to deal with user interface controls, in classes. The resulting TOOL source code now accompanies the system.

The source library contains basically three levels; classes in all levels can be used in an application. The low level maps in a straightaway fashion the Windows system. Actually, most methods in these classes are written in three lines, representing a test to see if the receiving object has been created, in which case a virtual machine function corresponding to an adequate Windows function is invoked. Classes in this level will rarely be needed in applications; they were written basically as a support for other levels. The intermediate level contains hierarchies representing visual objects, those appearing on user interfaces, and auxiliary objects, needed in the definitions of visual objects. Visual object styles and other classes are also in this level. These are classes providing application constructors with an organized view of the object oriented, graphical user interface world, while at the same time providing detailed functionality. The high level library contains classes that simplify application construction at the cost of reduced functionality.

Tha language TOOL has also been used in the programming of an interactive application constructor, which allows the assembly of applications in a visual fashion. For example, end user interfaces can be constructed by interactively selecting and placing visual objects such as buttons and edit boxes on a window. Facilities for using alignment grids, for selecting color and font, and for defining tab orders also exist. During the assembly of an interface, the handlers for those messages that interface objects may send can also de declared. At the end of the assembly, a corresponding TOOL source program is automatically

constructed by the system. Application objects not of a visual nature but useful in applications, such as alarms, bitmaps and tables, can also be introduced in programs through this interactive constructor.

Other uses of TOOL include the construction of an educational animation package, which will contain algorithms for sorting, interpolation, and numerical integration, for example; the construction of CASE tools for the construction of software design diagrams such as DFD's; and the construction of a facility for identifying map regions.

SUMMARY

TOOL is a programming system involving an object and event oriented language built for message passing and graphical user interface software. The main feature of this language is the ability programmers have of defining, in classes, the synchronous or the asynchronous behavior of objects. The asynchronous behavior is a necessity of user interface objects, like buttons and windows. In TOOL, however, this feature is extensible in the language, in the sense that even in applications not dealing directly with graphical user interfaces the programmer may decide to advantageously use message passing, instead of tying up the application with execution stack control data. As an example, consider the emission of reports related to a payroll application; since no results affecting the application are expected back from such operations, one may decide to send messages requesting that reports be printed out.

Another distinguishing feature of the language is the definition of polymorphism as an object attribute, chosen by programmers at declaration. Such objects may change classes at execution time, provided their new classes are in the object's family. This in fact slows down

execution, since at execution time the current class of a polymorphic object has to be found out so that proper operations are applicable to it. This cost is, however, paid by programmer's choice; non-polymorphic objects allow compilers efficient code generation and reside on the execution stack, being accessed with ease.

BIBLIOGRAPHY

1. C. Petzold, *Programming Windows*, Microsoft Press, Redmond, Wash, 1990.

2. Open Software Foundation, *OSF/Motif Style Guide*, 1990.

3. A. Golgberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Palo Alto, CA 1983.

4. SPA Sistemas Planejamento e Análise, *TOOL: the Language*, Rio de Janeiro, Apr 92.

5. S. Carvalho, 'TOOL: a Short Description', Monografias em Ciência da Computação, n. 25/92, Departamento de Informática, PUC/RJ, Rio de Janeiro, 1992.

6. B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

7. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Mass., 1986.