COM Support in BETA

Ole Lehrmann Madsen
The Danish National Centre for IT Research
Computer Science Department, Aarhus University
Åbogade 34, DK-8200 Århus N, Denmark
Ole.L.Madsen@{cit.dk,daimi.au.dk}
Tel.: +45 8942 5670, Fax: +45 8942 2443
May 24, 1999

1. Introduction

Component technologies based on binary units of independent production [Szy97] are some of the most important contributions to software architecture and reuse during recent years. Especially the COM technologies and the CORBA standard from the Object Management Group have contributed new and interesting principles for software architecture, and proven to be useful in practice.

In this paper ongoing work with component support in the BETA language is described. The overall motivation is to get experience with component technologies in order to obtain a better understanding of the underlying principles for building software with components. The project is part of an industrial research project [COT97] where one of the research themes includes reuse, software architecture, and components. In practice it seems hard for developers to design component-based software without being heavily tied to the actual technology such as COM or CORBA. There is apparently a need for design techniques and/or technologies that makes it possible to design component software independent of the actual technologies.

There exist a number of tools that provide various degree of COM support. Unfortunately the tools support different aspects of COM and some of the tools provide a lot of support behind the scene. It is therefore not always easy to get a clear picture of whether or not a given feature is due to COM or the tool itself. A subproject (not reported here) has been to perform an evaluation of a number of COM tools including Delphi, Visual Basic, Visual C++, and Visual J++[Elm98].

To be able to experiment with component technologies without being tied to a specific product, it was decided to implement COM support for BETA. Such a project is of course useful in itself as it opens the world of COM for all BETA users. COM support in BETA should fulfill the following goals:

1. A BETA program should be able to *import* and use available COM components on Windows (NT, 95 or 98) and other platforms supporting COM.
2. It should be possible to implement COM components in BETA and *export* those for use by other applications.
3. The implementation of COM support should be *reused* as much as possible for other component architectures such as CORBA.

One of the characteristics of COM is the ability for a component to support several interfaces, but few programming languages have good support for implementing several interfaces. A block-structured language with nested classes provides good support for this, and in this paper we show how to use nested classes for implementing several interfaces. Using nested classes is not just applicable for COM, but a technique that can be used for structuring the architecture of components and frameworks in general. Nested classes may often be used as a structured alternative to multiple inheritance.

2. COM levels

From the description in [Bro95] and [Rog97] COM appears to be a nice and simple model for binary components. The idea of accessing a component through an interface that is basically a call though a virtual dispatch table is interesting and so is the idea of components supporting several interfaces.

When it comes to practice, it turns out that the handling of parameters via COM interfaces is a big mess. We have not been able to find any clear definition of parameter types that are supported by COM including a definition of their semantics (call-by-value, call-by-reference, call-by-in-out, etc.) It seems that almost all parameter types from Visual Basic, C and other languages have to be supported. In order to improve understanding of COM, to simplify the implementation and to be able to reuse part of the implementation for other component models, three levels of the COM model have been identified:

- **BASICCOM.** The BASICCOM level is a simple binary component model based on interfaces in the form of calls through a virtual dispatch table. The parameter concept is simple and clean and the following parameter types are allowed:
    - Simple values like integer, boolean, char, and real. For such simple parameters a call by value semantics is used. I.e. the value of a simple parameter is copied at the point of a call.
    - References to COM objects. For COM references a call-by-reference-value semantics is used. This basically means that the value of the reference (pointer) is passed at the point of the call.

  The BASICCOM level captures the central part of the COM model and this level may also be the basis for supporting other component models, such as CORBA.
- **EXTERNALCOM.** At this level a number of predefined interfaces and functions are defined in order to communicate with the external environment like Windows. This includes the interface `IUnknown` with operations `QueryInterface`, `AddRef` and `Release`, functions such as `CoCreateInstance` and means for identification of interfaces and components, which for COM are the global unique identifiers.
- **FULLCOM.** As mentioned, the BASICCOM model is simple and clean, but reality is more complicated, primarily due to the parameter mess. At the FULLCOM level all parameter types known from real COM interfaces are supported. It has not been possible to obtain a specification of which parameters actually to support and what the exact semantic should be. Parameter support in the current prototype has been developed on a need-by-need basis. Whenever a new interface was going to be supported, new parameter types often showed up. Of course we hope that this will stabilize. As of today the following parameter types are included: struct-by-value, struct-by-reference, pointer to integer, at least 3 string types: pointer to 8-bit char (*char), pointer to 16-bit char (*wchar), pointer to 16-bit char with explicit length (BSTR), variant (union), safearray, and many more.

We will mainly discuss the BASICCOM level, since we are primarily interested in the basic principles of component technology.

3. Language Support

In this section we will at the language level describe how a BETA program may interface to COM. First it is shown how external COM components may be interfaced from BETA. Then it is shown how to implement COM components in BETA and export them to external languages. Finally it is shown how to support several interfaces using nested classes.

3.1 Importing COM to BETA

To access a COM component from BETA, an interface to the component must be described in BETA. Consider the following COM interface in IDL:

```
interface Ifoo
{ virtual int __stdcall foo1(int a, int b);
  virtual Ibar* __stdcall foo2(double r; Ibar *Y);
  virtual Itext* __stdcall foo3(char ch);
};
```

The corresponding BETA version looks as follows:

```
Ifoo: COM
  (# foo1:< (# a,b,c: @int32 enter(a,b)exit c #);
     foo2:< (# r: @real; Y: ^Ibar enter (r,Y[]) exit Y[] #);
     foo3:< (# ch: @char; w: ^Itext enter ch exit W[] #);
  #);
```

The interface `Ifoo` has three functions:

- `foo1` with two input parameters, `a`, `b` and an output parameter, `c`. All three parameters are integers and transferred as call-by-value.
- `foo2` with two input parameters, `r` and `Y`. The parameter `r` is of type real and call-by-value. The parameter `Y` is a reference (pointer) to `Ibar` interfaces. The parameter `Y` is also an output parameter.
- `foo3` with one input parameter, `ch` of class `char` and call-by-value and an output parameter `W` that is a reference to `Itext` objects.

In BETA a COM interface must be a subpattern of the predefined pattern `COM`. As can be seen, `Ifoo` is a subpattern of `COM`. The patterns `Ibar` and `Itext` are supposed to be subpatterns of `COM`.

The interface pattern `Ifoo` may be used in the following way:

```
main:
  (# S: ^Ifoo;
     n,m: @integer; x: @real;
     R: ^Ibar; T: ^Itext
  do 'IfooID' -> ImportObject -> S[];
     (n,n*m+12) -> S.foo1 -> n;
     (3.15,R[]) -> S.foo2 -> R[];
     '*' -> S.foo3 -> T[];
  #)
```

In pattern `main`, `Ifoo` is used to interface to an external component:

- `S` is a reference to instances (interfaces) of `Ifoo`.
- The function `ImportObject`[1] is assumed to return a reference to an `Ifoo` instance based on an id `'IfooId'`.
- `S` may be used as any other BETA reference to call virtual patterns as e.g. shown by the call `(n,n*m+12) -> S.foo1 -> n`.
- As can be seen only simple values and references to COM interfaces are passed as parameters to and from calls of COM interface functions.

The use of the words component, interface, class and object has been a bit imprecise in the above. In COM, *an interface is a specific memory structure containing an array of function pointers* [Rog97, Chap. 2, p. 15]. A COM component and its interfaces may then be implemented in OO languages using classes and objects.

[1] In practice a function like `CoCreateInstance` from the EXTERNALCOM level should be used.

In BETA there is a distinction between pattern and object where objects are instances of patterns. A pattern is a unification of class and procedure and other abstraction mechanisms. The term *class pattern* is often used for a pattern used as a class and similarly, the term *procedure pattern* is used for a pattern used as a procedure. To be consistent with this terminology, we will use the term *interface pattern* for a pattern used for describing an interface and *interface object* for an object that may be referred by a reference qualified by an interface pattern. An interface object may function as an interface to a component or be the real component itself. This depends on how the interface and corresponding class are implemented. We will return to this later. As we shall see later, an interface object contains a structure that is identical to a COM interface as defined in [Rog97]. We will use the words interface and interface object interchangeably as they in this context are isomorphic.

3.2 Exporting COM components from BETA

A COM component may be implemented in BETA as a subpattern of `COM`. The next example shows a simple calculator implemented as a COM object:

```
Calculator: COM
  (# clear:< (# do 0 -> sum #);
     add:< (# v: @integer enter v do sum + v -> sum #);
     subtract:< (# v: @integer enter v do sum - v -> sum #);
     result:< (# v: @integer do sum -> v exit v #);
     sum: @integer;
  #)
```

The interface defined by a subpattern of COM is the virtual pattern attributes of the pattern. In this example the interface is `clear`, `add`, `subtract`, and `result`. All other attributes are hidden from the (external) client. BETA supports virtual as well as non-virtual pattern attributes. Non-virtual pattern attributes and data-items are thus hidden from the client. The integer variable `sum` is an example of a hidden data-item. An instance of `Calculator` may be instantiated and exported as shown by the following example:

```
main:
  (#
  do 'CalculatorID' -> RegisterID;
     &Calculator[] -> ExportObject
  #)
```

The procedure patterns[2] `RegisterID` and `ExportObject` are supposed to be external functions that can register an interface identification and deliver the reference `S` to an external (COM-) component.

It is of course possible to define further subpatterns of `Calculator`:

```
ExtendedCalculator: Calculator
  (# multiply:< (# v: @integer enter v do sum * v -> sum #);
     divide:< (# v: @integer enter v do sum div v -> sum #)
  #)
```

The interface for `ExtendedCalculator` is then the interface for `Calculator` extended with `multiply` and `divide`.

In the above example, the interface and component were defined together as one pattern. It is of course possible to separate the definition of the interface and the pattern. An `ICalculator` interface may be defined and `Calculator` may then be implemented as a subpattern of `ICalculator` or using the technique described in the next section.

[2] In practice "real" functions from the EXTERNALCOM level should be used.

3.3 Components with several interfaces

The following examples show how to define a component with several interfaces using nested class patterns. This style has been used for many years in BETA as an alternative to multiple inheritance [MM93][3]. Nested classes do not cover all aspects of multiple inheritance, but they provide a nice structural organization of several interfaces to the same component. The interface patterns `printable` and `drawable` describes properties of objects to be printable or drawable:

```
printable:
  (# print:< (# do INNER #);
     medium:< (# S: ^stream enter S[] do INNER #);
  exit this(printable)[]
  #);
drawable:
  (# draw:< (# do INNER #);
     move:< (# dx,dy: @integer enter(dx,dy) do INNER #);
  exit this(drawable)[]
  #)
```

A printable object has two virtual functions `print` and `medium`. `Medium` is supposed to set the print medium to be some stream (screen, text, file, etc.). A drawable object can be drawn or moved to a new relative position. The exit parts of `printable` and `drawable` define the value of printable- and drawable objects to be a reference to the object itself, i.e. the self reference. This is a BETA technicality that ensures that instantiation is simple to express. We next describe a pattern `Person` that implements the printable and drawable interfaces.

```
Person:
  (# name: ^text;
     asPrintable: printable
       (# print::< (# do name[] -> currentStream.putText #);
          medium::< (# do S[] -> currentStream[] #);
       #);
     asDrawable: drawable(# ... #);
     currentStream: ^stream
  #)
```

A `Person` object consists of

- Two data-items, `name` and `currentStream`.
- A nested pattern `asPrintable` that is a subpattern of `printable`. The virtual procedure patterns of `printable` are extended in `asPrintable`. Because of the block structure, `print` and `medium` can access variables in the enclosing `Person` object. As can be seen `print` prints the name of the `Person` on the `currentStream` and `medium` sets the variable `currentStream` to a new value (errors are ignored).

`Person` may be used as in the next example:

```
main:
  (# Joe: ^Person; prt: ^printable; drw: ^drawable
  do &Person[] -> Joe[];
     Joe.asPrintable -> prt[];
     screen[] -> prt.medium;
     prt.print;
     Joe.asDrawable -> drw[];
     (111,112) -> drw.move;
     drw.draw
  #)
```

[3] In [MM93] the focus is on how to use part-objects as an alternative to multiple inheritance.

- A new instance of `Person` is assigned to `Joe`.
- The `printable` interface of `Joe` is accessed through the call `Joe.asPrintable` and assigned to the reference `prt`. The person object may then be accessed through the printable interface by calling the operations `screen[] -> prt.medium` and `prt.print`.
- The `drawable` interface is used in a similar way.

The nice thing about this style of programming based on interfaces is that from the point of use of e.g. the `printable` interface, nothing can be assumed about the actual objects being printed. I.e. `prt` may refer to a `printable` interface implemented by any object.

The use of nested classes to implement several interfaces can be used in general and is not specific to COM. We will not go into details about the specifics of COM but just indicate how several interfaces and `IUnknown` may be implemented.

```
IUnknown: COM
  (# QueryInterface:<
      (# id: ^text; IF: ^IUnknown
      enter id[] do inner
      exit IF[]
      #);
     AddRef:< (# do inner #);
     Release:< (# do inner #)
  #)
Person: IUnknown
  (# QueryInterface::<
       (#
       do (if true (* if no match then IF[] = NONE by default *)
           // 'printableId' -> id.equal then asPrintable -> IF[]
           // 'drawableID' -> id.equal then asDrawable -> IF[]
          if);
          inner
       #);
      AddRef::< (# ... #);
      Release::< (# ... #);
      asPrintable: printable(# ... #);
      asDrawable: drawable(# ... #);
      name: ^text;
      currentStream: ^stream;
  #)
```

To be COM compliant, at least the following modifications to the above example are needed: `QueryInterface` should return an `HRESULT` value, `printable` and `drawable` should inherit from `IUnknown`, and `QueryInterface` should return unique instances of `asPrintable` and `asDrawable`.

4. Implementation

The main change at the BETA language level is that interface patterns must be subpatterns of COM. The reason for this is that the calling sequence for a virtual in a COM pattern is treated in a special way. Consider the pattern `foo` describing a component implementing the interface `Ifoo`:

```
foo: Ifoo
  (# foo1::< (# ... do ... #);
     foo2::< (# ... do ... #);
     foo3::< (# ... do ... #);
  #);
```

Instances of `foo` are generated as any other BETA object. The virtual procedure patterns `foo1`, `foo2` and `foo3` are handled in a special way since their run-time organization and code layout must be in a form where they can be called from external languages. They can therefore only be used as virtual procedure patterns and not as general patterns. It is for instance not possible to use them as class patterns and generate instances from them.

The BETA garbage collector used to move objects around. COM objects cannot be moved, since references to COM objects may be given to external code. A change was therefore necessary here. The memory management of BETA is based on generational scavenging with a traditional mark-sweep in the old object space. A completely new object management scheme has been implemented. For the new object space, scavenging is still used, but in the old object space, free space is collected in sophisticated lists and objects are not moved. COM objects are allocated directly in the old object space to ensure that they are never moved. The new memory management system has also been designed to support persistent objects in a more direct way than the current persistent object store. For a further description of the new persistent store see [Kor99].

A BETA object has a reference to a prototype keeping run-time information such as the virtual dispatch table for the object. The prototype pointer used to point to the start of the prototype, but has been changed to point directly to the dispatch table. For the `Ifoo` example the new run-time layout is as shown in Figure 1. `S` is a reference qualified by the `Ifoo` interface and referring an instance of `foo`. The dashed lines illustrate the `foo`-object with the solid lines showing the reference to the dispatch table in the prototype for `foo`. In the old implementation, the prototype pointer in the `foo`-object pointed to the start of the prototype (the start of the dashed box in Figure 1).

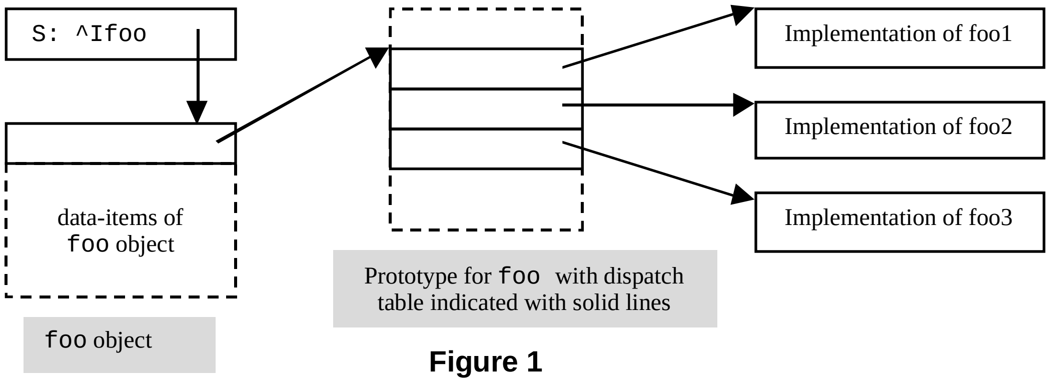

Figure 1

It was pretty straightforward to implement support for BASICCOM although it is always a time consuming task to implement and debug a new memory management system. To add the necessary interfaces and external functions to support EXTERNALCOM was also a straightforward task. The most time consuming and frustrating task has been to support FULLCOM with all the many different parameter types. As mentioned it turned out to be impossible to obtain a complete definition of COM IDL. The current implementation is thus based on an ad hoc extension of CORBA IDL that currently can handle all known IDL interfaces defined in Windows.

A translator from IDL to BETA has been implemented, which means that BETA interface patterns are available for all Windows interfaces. It has been a time consuming task to develop a suitable COM IDL and to implement the translator. To be able to support the many different parameter types, a special `holder` pattern has been implemented. The `holder` pattern is inspired by Microsoft J++ and can be used to represent pointers to integers, pointers to references to COM objects, etc. Due to space limitations no details will be given here.

Garbage collection. As mentioned above, the object management scheme for BETA has been changed to prevent COM objects to be moved around. There is, however, more to garbage collection than to prevent COM objects from being moved.

There is no direct support for garbage collection (`addRef` and `release`) in the current implementation. When a component is imported to BETA, the programmer must keep track of it and make an explicit release of it when it is no longer used in the BETA code. When a BETA object is collected by the garbage collector, it may contain references to COM objects. It is currently the responsibility of the programmer to make sure that a release-call is made on such COM references if necessary.

For COM components implemented in BETA, it is also the responsibility of the programmer to keep track of the object. When the component is created and exported, the BETA program should keep a reference to it in some global table to prevent it from being collected by the BETA garbage collector. The component should have an explicit reference counter to be manipulated by `addRef` and `release`. When the counter reaches zero, the possible global reference to the component may be removed and the component may then be collected by the BETA garbage collector.

Most COM tools handle `AddRef` and `Release` automatically, and we expect to add better support for this in a future release.

5. Status and future work

As of today a working prototype has been implemented and COM support is expected to be part of the next official release of the Mjølner System. The prototype is currently being used in a number of projects [Coc97, Des98, COT97] for developing COM-based applications and COM-based frameworks. To complete the BETA COM implementation, the following parts should be done:

· As mentioned above, the support for garbage collection should be improved.

· In the current implementation, the value of `HRESULT` must be checked explicitly by the client. We expect to add a higher level of exception handling.

· In the current implementation dynamic load (using DLL's) is not fully supported.

· The Mjølner System includes powerful tools such as browser, structure editor, CASE tool, interface builder and debugger. It is planned to provide better support for components based on input from e.g. [KP98, Elm98].

· It is planned to extend the COM implementation to support CORBA. We expect the BASICCOM layer to be unchanged from a programmer point of view. The changes will primarily be at the EXTERNALCOM layer where a similar layer for CORBA probably will be introduced.

· In [Elm98] a study of mainstream tools was made. A comparison with environments like Oberon and Eiffel needs to done.

Acknowledgements. The work described here is part of COT [COT97], which is an industrial research project sponsored by the Danish National Centre for IT Research. Peter Andersen, Morten Grouleff, Christian Heide Damm, Stephan Erbs Korsholm, and Lennert Sloth have made significant contributions to the project. In addition discussions with other participants of COT have been useful. The idea of implementing COM support for BETA was triggered by discussions with Clemens Szyperski and other members of the Oberon team. Peter Andersen, Christian Heide Damm, Henrik Røn, Lennert Sloth and Peter Ørbæk gave useful comments on a previous version of this paper.

6. References

[Bro95] Kraig Brockschmidt: *Inside OLE, 2nd edn*. Microsoft Press, Redmond, WA, 1995.

[Coc97] Project *Coconut: Collaboration & Components, (inter) Net-based Ubiquitous Teleworking*, `http://www.cit.dk/Cocunut`, 1997-99.

[COT97] Project *COT: Centre for Object Technology*, `http://www.cit.dk/COT`, 1997-2000.

[Des98] Project DESARTE: Computer-Supported Design of Artefacts & Spaces in Architecture and Landscape Architecture, `http://www.daimi.au.dk`, 1998-2000.

[Elm98] Réne Elmstrøm, Peter Petersen, Kåre Kjelstrøm, Morten Grouleff, Peter Andersen, Henrik Lykke Nielsen, Kristian Lippert, Steen D. Olsen, Kim Vestergaard: *Evaluation of COM Support in Development Environments*, Centre for Object Technology, Aarhus University, Technological Institute, COT/3-4, 1998, available from `http://www.cit.dk/COT`.

[Kor99] Stephan Korsholm: *Transparent, Scalable, Efficient, OO-persistence*, Submitted for ECOOP'99 workshop on object-oriented databases.

[KP98] Kaare Kjelstrøm, Peter Petersen: *A CASE Tool for COM Development*, The Eighth Nordic Workshop on Programming Environment Research 1998, NWPER'98. Revised version in Nordic Journal of Computing, 6(1), Spring 1999.

[MM93] Ole Lehrmann Madsen, Birger Møller-Pedersen: *Part Objects and their Locations*. In Proc. Technology of Object-Oriented Languages and Systems – TOOLS'10, (B. Magnusson, B. Meyer, J.F. Perrot, eds.), Prentice-Hall, 1993, pp. 283-297.

[Rog97] Dale Rogerson: *Inside COM*, Microsoft Press, Redmond, WA, 1997.

[Szy97] Clemens Szyperski: *Component Software - Beyond Object-Oriented Programming*, ACM Press/Addison Wesley, 1997.