Joint NCC & IIS Bull., Comp. Science, 8 (1998), 39–56
© 1998 NCC Publisher

XDS-COM — a COM binding for Oberon-2 and Modula-2

Timofei Kouzminov

The XDS-COM toolkit provides a language binding of the Microsoft COM to Oberon-2/Modula-2. It can be used to develop both COM clients and servers. Multiple interfaces of a single object are created in a natural way. The XDS-COM does not introduce any language extensions but extensively uses the rich run-time support of the XDS development system.

1. Introduction

XDS is a portable Oberon-2/Modula-2 development system of the XDS company in Novosibirsk, Russia [1]. It produces native code for Win32, OS/2 and various flavors of UNIX and also can convert Oberon-2/Modula-2 programs into C.

The Component Object Model (COM) [2-4] is a mechanism provided by Microsoft to perform object-oriented communication between objects in a multi-process distributed environment. COM is a binary standard describing data structures (interfaces) plus some API. Microsoft provides a language binding of COM to C and C++ and automates some routine COM work in their class library, MFC.

This is also a target of XDS-COM toolkit — to provide a language binding to Modula-2 and Oberon-2 and create a number of classes to facilitate writing of COM clients and servers.

A solution to this problem is provided by the Direct-To-COM compiler by Oberon Microsystems [5, 6]. This compiler provides a set of COM-oriented language extensions including a pseudo-module `COM`. The presence of multiple interfaces of the same object is not supported directly by Direct-To-COM, though it can be modelled by a linked structure of objects acting as a single COM object.

Here we present an alternative approach. We do not make any specific COM language extensions. Instead, we use the rich run-time support of XDS including access to type descriptors and take advantage of interoperability with C language already provided by XDS. The main feature of our solution is, that despite only single inheritance being permitted in Oberon-2, a server object using XDS-COM may have multiple interfaces.

At present the core parts of XDS-COM are implemented and we have a working example of COM client and server both written in Oberon-2.

The overall structure of the paper is as follows:

- The rest of this section briefly overviews the main notions of COM.
- The **first problem** which is solved in XDS-COM is to provide convenient access to COM objects (including COM objects implemented by non-XDS applications) from XDS programs. Sections 2 and 3 describe the solution to this problem.
- Section 4 describes the solution to the **second problem** — how to implement COM objects in XDS.
- Section 5 outlines yet **unsolved problems** and proposes further directions of the work.

1.1. Interfaces and interface pointers

Objects in COM are accessible via *interface pointers*. An interface pointer (IP) is a pointer to a pointer to a method table. The method table consists of pointers to functions (values of procedure type) each of which takes an interface pointer as its first argument. The names, types of arguments, order and semantics of methods in the method table are determined by the *interface* — a contract between a method caller (client) and method implementation provider (server).

Each interface assigned an *interface identifier* (IID) which is actually a 128-bit *globally unique identifier* (GUID). Microsoft claims that their technology allows the creation of GUIDs that are different from any GUID created elsewhere.

Interfaces support single inheritance, that means that a method table of a derived interface has at the beginning all the methods with the same names and arguments as methods of a base interface in the same order. Unlike inheritance in Oberon-2 or C++, interface inheritance is **not** a regular method to enrich the capabilities of an object, because this is achieved by adding **multiple interfaces** to the same object.

Each interface inherits interface `IUnknown` which provides the basic capabilities of COM objects. Among them is the `QueryInterface` method which is used to receive another interface of the same object given an IID.

As mentioned above, a COM *server* is an executable (`.exe` or `.dll`) which implements COM objects and a COM *client* is an executable which invokes the methods of COM objects. As in Oberon-2, we say that a COM server *exports* an IP, and a COM client *imports* it.

The operating system supports *marshalling* of IPs, that is taking an IP in the address space of an application and creating the corresponding *proxy*

IP in the address space of another application. Method calls to a proxy IP are automatically routed to the server.

The DCOM (Distributed COM) feature allows marshalling of IPs through a network. It makes invocation of methods on a remote server transparent.

2. Access to COM objects

2.1. Representation of COM interfaces

In XDS-COM each *interface pointer* is a "C" pointer to "C" pointer to a method table. A set of methods in the method table is specific for a COM interface.

The method tables are represented in XDS-COM as records inheriting empty record `objBase.VTBL`. This record is an `"Oberon"` record to allow inheritance but you should never use

```
POINTER ["Oberon"] TO VTBL.
```

As every interface inherits the basic COM interface `IUnknown`, the method tables of other interfaces should inherit `objBase.IUnknown_VTBL`, not `VTBL` itself.

Declaration of an interface `IFoo` inheriting `IBar` causes declaration of the following entities:

- The constant `IID_IFoo` of type `com.GUID` represents the unique *interface identifier*. This identifier may be obtained by the **guidgen** utility from Win32 SDK or by the `CoCreateGuid` API function.
- The `"Oberon"` *method table* record type `IFoo_VTBL` inheriting `IBar_VTBL` and augmenting it with methods (**fields** of procedure type)

  ```
  method: PROCEDURE [com.STDMETHOD] (com.ipIFoo, ...).
  ```

  The order of methods is significant.
- The `pcIFoo_VTBL` type which is

  ```
  POINTER "C" TO IFoo_VTBL.
  ```

- The *interface pointer* (IP) type `ipIFoo` which is

  ```
  POINTER "C" TO IFoo_VTBL.
  ```

- Conversion functions `IP_IFoo` and `IFoo_IP` which convert from ipIUnknown to `ipIFoo` and back. These functions are added only for convenience, their implementations consist of single calls of `SYSTEM.VAL`.

Declarations of these entities may be (but still are not) generated automatically from IDL definition or COM type library (`.tlb`). Having the entities described above, it is possible to invoke methods of an *imported* COM interface pointer.

For an IP named `ip`, the call syntax is

```
ip^^.method(ip, ...).                                   (1)
```

Certainly, the caller must follow COM rules of reference counting. That means that each time an IP is assigned to another variable, the `AddRef` method must be called:

```
ip1 := ip;
ip^^.AddRef(ip);
```

And each time an IP value is lost, `Release` must be called:

```
ip^^.Release(ip);
ip := NIL;
```

2.2. Interface objects

To provide good syntax for method calling and to automate reference counting we introduce Oberon-2 *interface objects* (records inheriting `com.INTERFACE`, the corresponding pointer type is `com.pINTERFACE`).

For every COM interface `IFoo` inheriting `IBar` the following entities are declared:

- The record type `IFoo` (*interface object*, IO type), which inherits `IBar` but does **not** add any record fields. All such record types inherit (maybe indirectly) `com.IUnknown` which, in turn, inherits `com.INTERFACE`. `IFoo` adds its methods to `IBar` in the form:

  ```
  PROCEDURE [com.STDMETHOD] (p: pIFoo) method(...).
  ```

  Note that the declarations of the methods must be **in reverse order** because of the XDS compiler feature. It is possible to use forward method declarations to specify the order of methods before actual declarations.

- The *interface object pointer* (IOP) type

  ```
  pIFoo = POINTER "Oberon" TO IFoo.
  ```

- The variable

  ```
  dIFoo: com.pITF_INFO,
  ```

which is a pointer to the *interface descriptor* (typed `com.ITF_INFO`). The interface descriptor must be created and initialized with its `Init` method. Interface descriptors are used to link interface identifiers and Oberon-2 types at run-time.

The **first** field `vtbl` of an IO is a `"C"` pointer to method table. For each IO this field is initialized (by the `Init` method) with a pointer **into the method table** of `IFoo` in such a way that it points to the first method not belonging to `com.INTERFACE` (actually it is the `QueryInterface` method of `IUnknown`).

So for an IO, we can make calls in convenient style:

```
    p.method(...).                                          (2)
```

At the same time, a pointer to IO may be used as an IP if a conversion is provided. In this case it may be used as a COM interface pointer from any language, for instance like (1).

This allows the **export** of interface pointers from XDS programs. If both the client and server are the same program, the (2) call form is preferable. Remember that you **never** need to call `AddRef` or `Release` methods of IOs.

Another field of every IO is `info: com.pITF_INFO` which points to the interface descriptor of specific interface. This allows the use of an IO to query for interface identifier and type descriptor of the Oberon-2 type implementing the interface.

The values of both `vtbl` and `info` fields are set by the `Init` method which must be called for every IO. Note that if an IO is created by a function from the XDS-COM library, it is initialized automatically.

The variable `dIFoo` needs initialization (`NEW` and `Init` method call). This may be done in the initialization part of module which declares `IFoo`.

2.3. GUID search table

XDS-COM uses a hash table mapping GUIDs to `comUtil.EL` records. This table is implemented by the `comUtil` module. As the `com.ITF_INFO` type inherits `comUtil.EL`, interface descriptors can be (and actually are) included into the hash table. It allows one to find the interface descriptor the and run-time type information of interface object type given an interface identifier.

2.4. Example

We declare the `IqHello` interface with 3 methods — get/put a hello message and output the message on the console. We create an Oberon-2 module `qHello` and declare the needed entities:

```
CONST
  IID_IqHello* = com.GUID{0E8A9D0E8H, 20F9H, 11D0H,
                    com.GUID_ARR{0A5H, 003H, 008H, 000H,
                                 02BH, 0E6H, 0C7H, 056H}};
TYPE
  pcIqHello_VTBL = POINTER ["C"] TO IqHello_VTBL;
  ipIqHello* = POINTER ["C"] TO pcIqHello_VTBL;
  IqHello_VTBL* = RECORD (com.IUnknown_VTBL)
    put_HelloMessage : PROCEDURE [com.STDMETHOD] (
      this : ipIqHello;
      (*[in] *) msg : BSTR
    (* BSTR is "Unicode string with length" type which
       is standard to COM *)
    ) : com.HRESULT;
    get_HelloMessage : PROCEDURE [com.STDMETHOD] (
      this : ipIqHello;
      (*[out, retval] *) VAR msg : BSTR
    ) : com.HRESULT;
    SayHello : PROCEDURE  [com.STDMETHOD] (
      this : ipIqHello
    ) : com.HRESULT;
  END;
PROCEDURE IP_IqHello*(ip: com.ipIUnknown): ipIqHello;
BEGIN
  RETURN SYSTEM.VAL(ipIqHello, ip);
END IP_IqHello;
PROCEDURE IqHello_IP*(i: ipIqHello): com.ipIUnknown;
BEGIN
  RETURN SYSTEM.VAL(com.ipIUnknown, i);
END IqHello_IP;
VAR dIqHello* : com.pITF_INFO;
```

In qHello we also declare a class identifier (see Subsection 4.2) of the class of HELLO objects:

```
CONST
CLSID_qHello* = com.GUID{0E8A9D0EBH, 20F9H, 11D0H,
                  com.GUID_ARR{0A5H, 003H, 008H, 000H,
                               02BH, 0E6H, 0C7H, 056H}};
```

Now we proceed to declarations of IO type:

```
TYPE
  IqHello* = RECORD (com.IUnknown) END;
  pIqHello* = POINTER TO IqHello;
```

```
(* Forward declaration to ensure proper (reverse) order
   of methods *)
PROCEDURE [com.STDMETHOD] ^ (this : pIqHello)
                SayHello*() : com.HRESULT;
PROCEDURE [com.STDMETHOD] ^ (this : pIqHello)
                get_HelloMessage*(
                  (*[out, retval] *) VAR msg : BSTR
                ) : com.HRESULT;
PROCEDURE [com.STDMETHOD] ^ (this : pIqHello)
                put_HelloMessage*(
                  (*[in] *) msg : BSTR
                ) : com.HRESULT;
```

Here we declare the basic variants of `IqHello` methods to allow the `IqHello` type serve as an IHO (see Subsection 3.1):

```
PROCEDURE [com.STDMETHOD] ^ (this : pIqHello)
                put_HelloMessage*(
                  (*[in] *) msg : BSTR
                ) : com.HRESULT;
VAR p : ipIqHello;
BEGIN
  p := SYSTEM.VAL(ipIqHello, this.ip);
  IF p <> NIL THEN
    RETURN p^^.put_HelloMessage(p, msg);
  END
END put_HelloMessage;
PROCEDURE [com.STDMETHOD] ^ (this : pIqHello)
                get_HelloMessage*(
                  (*[out, retval] *) VAR msg : BSTR
                ) : com.HRESULT;
BEGIN
  p := SYSTEM.VAL(ipIqHello, this.ip);
  IF p <> NIL THEN
    RETURN p^^.get_HelloMessage(p, msg);
  END
END get_HelloMessage;
PROCEDURE [com.STDMETHOD] ^ (this : pIqHello)
                SayHello*() : com.HRESULT;
BEGIN
  p := SYSTEM.VAL(ipIqHello, this.ip);
  IF p <> NIL THEN
    RETURN p^^.SayHello();
  END
```

```
END SayHello;
```

At last we proceed to the initialisation of interface descriptor:

```
CONST ThisModule = "qHello";
BEGIN
  NEW(dIqHello);
  dIqHello.Init(ThisModule, "IqHello", IID_IqHello);
END qHello.
```

3. COM clients

3.1. Interface handle objects

XDS-COM clients may use imported interface pointers through *interface handle objects* (IHO) — a subset of interface objects which have an interface pointer assigned to an `ip: ipIUnknown` field defined for every IO. This is because methods which are generated for each IO type simply redirect the calls to the `ip` if it is not NIL.

That means that each (2) call to an IHO is redirected to a (1) call to the corresponding IP.

An IHO for a given IP is created by the procedure `comCli.Instance`. This procedure creates and initializes an interface object of the appropriate type, assigns the value to the `ip` field and sets a finalizer to perform `Release` on the IP when the IO is destroyed.

Note that `AddRef` is not performed on the IP pointer. The logic behind this is that after you called

```
      p := comCli.Instance(ip, dIFoo)
```

you do not need the `ip` value any more. The first parameter of `comCli.Instance` is actually a `VAR` parameter and `NIL` will be assigned to it on the function return. So the total number of references to an object will not change.

The procedure `comCli.Instance` is an examle of usage of XDS run-time support. It is essential that new objects may be created while the type of these objects is specified is not known at compile-time.

3.2. Other client utilities

The `comCli` module provides helper procedures for standard COM mechanisms to **import** IPs — search for a class factory, search for an object of a given class etc. The difference between `comCli` procedures and standard COM API calls is that `comCli` utilities accept pointers to interface descriptors, not interface identifiers and return IO pointers, not IPs.

3.3. Example

The client module qHello_c imports qHello. In this module we can create and use an IHO of a HELLO object:

```
VAR
  r: com.HRESULT;
  itf: com.pIUnknown;
  h: qHello.ipIqHello;
  s: com.BSTR;
   .  .  .
r := comCli.CreateInstance(
     qHello.CLSID_qHello, (* Object class *)
     NIL,
     objBase.CLSCTX_ALL,
     NIL, (* A remote server may be specified here *)
     qHello.qIqHello, (* Interface needed *)
     itf (* Put result here *)
);
h := itf(qHello.pIqHello);
s := comUtil.ToBStr("Hello, Sun!");
h.put_HelloMessage(s);
comUtil.FreeBStr(s);
h.SayHello();
```

4. COM servers

4.1. Performing objects

Oberon-2 objects that wish to export COM interfaces must be implemented as *performing objects* (PO) of type inheriting com.OREC. The notion of PO does not belong to COM but rather to COM implementation in XDS.

If you wish to implement a COM interface to an object which must reside in some other place in type hierarchy, the usual solution can be applied. Just create an intermediate PO type inheriting com.OREC and place a pointer to the actual object type somewhere in the PO.

All POs must be linked in a **cyclic** list of all objects living at a particular server. Usually there is only one PO list in an .exe or .dll COM server. POs are included in the list when being initialized by the Init method, an object already in the list must be specified as an argument of this method.

The basic PO type com.OREC implements QueryInterface, AddRef and Release methods as needed for the IUnknown interface. AddRef and Release use a ref_count field which is kept equal to the total number of references to this object through IPs (ipIFoo) both from inside and from outside the application. When this number reaches zero as the result of the `Release` method, the PO is excluded from the list. If there are no other references to the object, it is a subject for garbage collecting.

4.1.1. Interface lists for POs

Every PO has the field `itf` which points to the list of *interface list objects*. The type of each interface list object corresponds to an interface of PO and inherits some interface object type, i.e., `IFoo`. But only for `IUnknown` interface this type is `IUnknown` itself. For any other interface `IFoo` of a PO of type `OBJ` the type of the interface list object is `IFoo_i_OBJ`.

`IFoo_i_OBJ` inherits `IFoo` but does not add any fields or methods. It simply redefines the methods of `IFoo` (including the methods of any inherited interface object type `IBar` except `IUnknown`) to redirect calls to corresponding methods of `OBJ`.

To provide this each IO has the field `orec: com.pOREC` which is used only by interface list objects and points to the PO owning the list.

The default implementation of `QueryInterface` for PO scans the interface list objects from the beginning checking whether the interface inherits needed interface. So if the `IUnknown` interface is queried, the same interface list object will always be found each time, just as COM requires.

The methods of `IFoo_i_OBJ` can be generated automatically if OBJ has the methods with the same names and argument types as IFoo. But sometimes it is useful to modify automatically generated methods for instance to perform some argument conversions. Some useful conversions are presented in the following table:

Arguments of IO methods	Arguments of PO methods
interface identifier (`com.GUID`)	interface descriptor (`com.pITF_INFO`)
interface pointers (IP)	interface object (IO) pointers
Unicode strings (`wTypes.OLESTR`, `objBase.BSTR`)	character arrays
Module `comUtil` provides necessary utilities for string conversion	

Interface list objects must be created for each PO after PO creation and initialization. This is done by repeatedly calling the `AddInterface` method each time specifying a different interface descriptor as the argument. This method uses the XDS facility of finding object types by name and creating new objects using a run-time reference to type descriptor.

For objects which are created by class factories provided by the XDS-COM interface lists are created automatically.

4.1.2. Lock counting

Each PO has the method `Locks` returning the number of external references to this object. For ordinary POs the value returned is equal to the reference count of the object, but `Locks` may be redefined for other PO classes. The `Release` method actually checks the `Locks` result and excludes a PO from the list only if it is zero.

4.2. Class factories

In COM, the notion of *class* is replaced by the notion of *class factory*. A class factory is an object with interface `IClassFactory` which provides the `CreateInstance` method of creating new objects. Class factories are registered by their unique (`com.GUID`) *class identifiers*.

COM does not fix any implementation of class factory. XDS-COM provides two standard PO types to implement class factories:

- single-use class factory `comSrv.CF_SINGLEUSE`;
- multiple-use class factory `comSrv.CF_MULTIPLEUSE`. "Multi-separate" class factories introduced by COM are implemented as a particular case of multiple-use ones.

Class factories of both types are created by corresponding methods of a server object (see Subsection 4.5).

Class factory POs (CFs) are included both in the cyclic PO list and in the hash table allowing to find a CF with the class identifier.

Just according to COM rules, a single-use CF is linked to a previously created PO and always returns this very PO as the result of `CreateInstance`.

A multiple-use CF remembers the `type` of a PO and the list of necessary interfaces. Each time `CreateInstance` is invoked, it creates a new object of this type and supplies it with interface list objects.

4.2.1. Registration of class factories

A class factory may be registered with COM using API call `CoRegisterClassObject`. When CFs are registered, the registration handle returned by API call is stored in the `reg` field to provide unregistering.

As registration automatically increases the reference counter by one, CF redefines the `Locks` method in such a way that the reference introduced by registration is not counted. At the same time additional locks introduced by the `LockServer` method of the `IClassFactory` interface are counted. So a CF may be released if it has no external references except the one introduced by registration and no additional locks.

For every CF a finalizer is provided which unregisters the class factory if it was registered.

4.3. Containment and aggregation

As there is no implementation inheritance in COM, two alternative methods are used to provide the object code re-use. In both methods an object called *outer* object contains among its data a reference to an *inner* object.

The first method is called *containment.* When this method is used, some methods of the outer object are implemented using calls to methods of the inner object. XDS-COM does not provide any special means to implement containment as it can be done in a straightforward way.

Another method for the code re-use is called *aggregation.* When this method is used, we say that the outer object aggregates the inner one. When the outer object is queried for an interface possessed by the inner object, it can return an IP of the inner object. In fact the inner object impersonates the outer one in this particular interface. That means that all methods of `IUnknown` inherited by this interface must be delegated to corresponding methods of the **outer** object.

COM itself provides no special tools to implement aggregation except when an object which must be aggregated is created by the `CreateInstance` method of a class factory, an additional parameter specifies the IP of the outer object.

XDS-COM automates aggregation and allows the objects implemented using XDS-COM to be aggregated by and to aggregate objects implemented without XDS-COM, for instance in C/C++. That means that all links between inner and outer objects must be IPs, not pointers to POs.

Each PO has a list of IPs of inner objects aggregated by it. For every object in this list two interfaces `base` and `derived` are specified. Each inner object is responsible (queried by the `QueryInterface` method of the outer object) for the interfaces which inherit `base` and are inherited by `derived`.

When an inner object is added to this list its lock count is increased by calling `AddRef` method of `IUnknown` interface. When an inner object is queried for an interface provided by this object its `QueryInterface` method is called. That means that `IUnknown` methods of the IP which is included in the list of inner object must not be delegated to the outer object but handled by the object itself. Such IPs are called *controlling* IPs and are marked by a flag in an interface object.

If an object is aggregated, an IP to the outer object is stored in the inner PO. When this IP is not NIL, the calls to the `IUnknown` methods of all non-controlling interfaces of the object are delegated to the outer object.

By default, only interface objects created for `IUnknown` interface are marked as controlling ones. So the common practice is to implement aggregated objects is to provide two interfaces: controlling `IUnknown` and non-controlling interface with the methods specific for this object.

The procedure `com.Aggregate` may be used to create links between existing inner and outer objects. But the preferred way to create aggregated objects is to let the class factories do it. A single-use class factory checks if the object being accessed must be aggregated and calls `Aggregate` if necessary.

When a multiple-use class factory is initialized a list of **class identifiers** of inner objects accompanied with `base` and `derived` specifiers may be created. For every new object created by the class facrory a set of appropriate inner object is created.

4.4. Non-standard object implementation

Certainly, anybody can use any other schemes of object implementation. It is possible to have a linked structure of objects instead of a single PO implementing different interfaces. It is possible to implement your own class factories. XDS-COM allows this but provides a simple scheme described above which guarantees that the somewhat complex COM rules are obeyed.

4.5. Server objects

A COM server is an executable program or a DLL. In both cases we need a special object (usually only one in a server) to implement some functionality of the server itself. This object is named server object, has the type `comSrv.SERVER` and is actually a PO. So a server object is included in the same cyclic list as all other POs. This allows the use of server object as a reference to the PO list when creating new POs.

4.5.1. In-process servers

The in-process server in COM is a DLL which is invoked when an object of particular class is needed. Objects implemented by an in-process server exist in the same address space as the client application.

Each in-process server must provide two functions: `DllGetClassObject` — to obtain class factory for a given class identifier and `DllCanUnloadNow` — to query whether there are no objects in use and the DLL may be unloaded. Both functions are implemented using methods of the corresponding server object `comSrv.Server`.

`DllCanUnloadNow` calls the `Locks` method of all POs in the cyclic list and returns `FALSE` if any object answered with a positive value. `DllGetClassObject` uses the hash table to find the CF for a given class identifier.

4.5.2. Local and remote servers

Both local (run on the same computer but as a separate application) and remote (run on another computer) COM servers are implemented as Windows executables. There is no difference between local and remote servers from the implementation point of view, all remoting stuff is transparently handled by the operating system.

XDS-COM provides necessary functionality of executable COM servers through methods of server an object. This functionality includes:

- Registering the server as a COM application when run with the switch /RegServer.
- Registering and unregistering class factories.
- Executing the message loop and stopping it when it is time to unload the server (method Serve).

If an executable server is run with an /Embedding switch (as it is run automatically by COM), it is unloaded when all its objects report Locks()=0. If the server is run manually, without the switch, one extra reference is counted for the server object itself thus avoiding automatic unloading.

4.6. Example

Module qHello_s implements an executable (local or remote) COM server. First, it declares a PO class HELLO inheriting com.OREC, initialisation methods for this class and methods implementing interface functions:

```
TYPE
  HELLO* = RECORD (com.OREC)
    msg : BSTR;
  END;
  pHELLO* = POINTER TO HELLO;
PROCEDURE (this : pHELLO) Init*(list: com.pOREC);
 BEGIN
  this.Init^(list);
  this.msg := comUtil.ToBStr("Hello, world!");
END Init;
PROCEDURE (VAR this : HELLO) put_HelloMessage*(
    (*[in]*) msg : BSTR
  ) : com.HRESULT;
BEGIN
  OleAuto.SysReAllocString(this.msg, msg);
  RETURN Windows.S_OK;
END put_HelloMessage;
PROCEDURE (VAR this : HELLO) get_HelloMessage*(
    (*[out, retval] *) VAR msg : BSTR
  ) : com.HRESULT;
BEGIN
```

```
  msg := OleAuto.SysAllocString(this.msg);
  RETURN Windows.S_OK;
END get_HelloMessage;
PROCEDURE (VAR this : HELLO) SayHello*() : com.HRESULT;
BEGIN
  comUtil.PrintBStr(this.msg);
  RETURN Windows.S_OK;
END SayHello;
```

Then, a class for interface list objects is declared with all needed methods.

```
TYPE
  IqHello_i_HELLO* = RECORD (qHello.IqHello) END;
  pIqHello_i_HELLO* = POINTER TO IqHello_i_HELLO;
PROCEDURE [com.STDMETHOD] (this : pIqHello_i_HELLO)
put_HelloMessage*(
    (*[in] *) msg : BSTR
  ) : com.HRESULT;
BEGIN
  RETURN this.orec(pHELLO).put_HelloMessage(msg);
END put_HelloMessage;
PROCEDURE [com.STDMETHOD] (this : pIqHello_i_HELLO)
get_HelloMessage*(
    (*[out, retval] *) VAR msg : BSTR
  ) : com.HRESULT;
BEGIN
  RETURN this.orec(pHELLO).get_HelloMessage(msg);
END get_HelloMessage;
PROCEDURE [com.STDMETHOD] (this : pIqHello_i_HELLO)
SayHello*() : com.HRESULT;
BEGIN
  RETURN this.orec(pHELLO).SayHello();
END SayHello;
```

Then, we create a CF object for HELLO class:

```
VAR
  cf : comSrv.pCF_MULTIPLEUSE;
CONST
  ThisModule = "qHello_s";
BEGIN
  g_CF := comSrv.Server.NewCF_MULTIPLEUSE(
    qHello.CLSID_qHello, (* CLSID of objects to produce *)
    ThisModule, "HELLO", (* Name of PO type *)
    "", "", "", "", (* Versioning info *)
    FALSE (* Not a multy-separate server *)
  );
  g_CF.WillHaveInterface(qHello.dIqHello);
```

At last, the server is run:

```
comSrv.Server.Serve;
```

5. Problems to solve

5.1. Technical problems

5.1.1. Interface naming

When specific interfaces are passed as parameters to XDS-COM utilities, pointers to interface descriptors are used to designate interfaces. At the same time, an IO type is created for every interface in use. It seems logical to place a pointer to an interface descriptor into type descriptor provided by XDS run-time support for an IO type, as there already is a reserved field.

But this is not enough because we need access to run-time interface type information even if there are no IOs of this type. A language extension would be convenient which allows the use of names of object **types** as constants of "type descriptor" type (`oberonRTS.Type`). This will allow one to get rid of interface descriptor names such as `dIFoo` and use type names like `IFoo` instead.

5.1.2. Threading

At present XDS-COM is used in single-threaded COM servers. We need more investigations about using it with different COM threading models — namely *apartment* threading model with multiple threads each having an isolate set of objects and *free* threading model where every object must maintain its integrity while being used by multiple threads and no message dispatch loop is needed.

5.2. Automatic generation

5.2.1. Automatic generation of interface definitions

Microsoft implementation of COM provides a language to define COM interfaces. This language is an extension of IDL, a well-known language used to define RPC and CORBA interfaces. The Microsoft `midl` translator converts IDL definition into a C/C++ header file. All standard COM interfaces are defined in IDL and corresponding headers to be included in `Windows.h` are generated by `midl`.

There is another language to define COM interfaces — the binary language of *type libraries* (`.tlb` files). In Microsoft implementation, this language is interoperable with IDL: a type library may be generated from an IDL file which, in turn, may import definitions from a type library.

From another side, IDL is not an essential part of COM and some implementations can do without it. For instance Delphi takes a declaration of an Object Pascal object and constructs all necessary COM stuff.

There are the following methods to provide necessary functionality in XDS-COM:

- Not to use IDL at all. Instead, modify the compiler to allow specification of more COM-specific details in declarations of IO types, methods and method arguments, for instance in brackets, like

  ```
  TYPE IFoo = RECORD ["uuid(...)", "hidden", ...]
                 . . .
              END;
  ```

  Generate type libraries using compiler output or dynamically, using run-time type information. In the latter case, this information must be more detailed and include method names and parameter types. Generation of IDL may also be needed to generate to provide exporting custom COM interfaces to programs written in C/C++.
- To write a converter from IDL into Oberon-2 definitions or directly into sym-files, analogous to `midl`. To become free of Microsoft utilities, provide generation of type libraries from IDL.

5.2.2. Marshalling

Using local and remote servers in COM requires marshalling — converting method calls into a format suitable to transfer between address spaces. Standard *marshalling* is performed by so-called proxy/stubs DLLs. The C/C++ sources of such DLLs are generated by midl from IDL definitions. At present, XDS-COM implementation can use proxy/stub DLLs generated by `midl` and compiled by Visual C.

In XDS-COM we should either generate Modula-2/Oberon-2 code from interface definitions or provide an unified *proxy/stub* DLL which uses run-time type information to accomplish its task. In the latter case run-time type information must be significantly enriched.

6. Conclusion

Even not mentioning distributed programming, implementation of COM is necessary for any Windows-targeted development system because Windowscompatible programs should support OLE and more API functions are implemented using COM (for example DirectX).

XDS run-time environment appeared to be suitable for COM implementation because of the following features:

- XDS supports static and partially run-time type information which allows to automate many of the COM functions. In particular, the possibility to assign a **type** value to a variable and use this value for typechecks and object creation is used in generic implementation of `IUnknown` and `IClassFactory` interfaces.
- XDS fully supports dynamic linking and DLL creation, including initialization and finalization code. This allows to create in-process COM servers and compile Oberon-2 interface definitions into DLLs used both by client and server.
- XDS allows direct specification of type attributes. That means, in particular, that a pointer in a Oberon-2 program can be marked as `"C"` one to circumvent garbage collection.

At the same time, proper implementation of COM under XDS requires creation of additional tools analogous to Microsoft `midl` compiler. This is the topic of further work.

References

[1] *XDS On-Line Documentation*, `http://www.xds.ru/xds`.

[2] Kraig Brockschmidt. *What OLE Is Really About*, Microsoft Corporation, July 1996,
`http://www.microsoft.com/oledev/olecom/aboutole.htm`.

[3] *The Component Object Model Specification (draft)*, Microsoft Corporation, October 1995,
`http://www.microsoft.com/oledev/olecom/title.htm`.

[4] *OLE Programmer's reference*, Microsoft Win32 SDK Online Books.

[5] *Direct-To-COM Compiler. Technical Documentation and Overview*, Oberon microsystems, Inc.,
`http://www.oberon.ch/prod/dtc`$_d$*ocu.html*.

[6] *Implementing COM Objects with the Direct-To-COM Compiler*, The Oberon Tribune, Vol 2, No 1, January 1997.