

# Direct-To-COM Compiler Provides Garbage Collection for COM Objects

Dominik Gruntz, gruntz@oberon.ch  
Beat Heeb, heeb@oberon.ch  
Oberon microsystems, Inc.  
Technoparkstrasse 1, CH 8005 Zürich, Switzerland

## Abstract

Microsoft's Component Object Model (COM) is a language and compiler independent interface standard for object interoperability on the binary level. Besides the binary layout, this standard also specifies how the lifetime of objects is controlled. As (manual) garbage collection mechanism reference counting is used. A Direct-To-COM (DTC) compiler is a compiler which directly supports COM. In this article we present a DTC compiler for the programming language Component Pascal that automates memory management. Such an integration of the COM standard directly into the compiler is a novelty. For programmers, the integration of COM into a strongly typed, garbage-collected language has two major advantages. Firstly, it brings all the amenities of automatic garbage collection to COM. Secondly, it makes the handling of COM interfaces typesafe.

This article focuses on automatic garbage collection. It first recapitulates the COM rules for reference counting and the problems they introduce, then presents techniques to overcome the difficulties with reference counting, and finally compares these techniques with the DTC compiler.

**Keywords:** COM, reference counting, garbage collection, smart pointers, Direct-To-COM, Component Pascal

## 1. Introduction

Microsoft's Component Object Model (COM) [Brockschmidt95,Rogerson97] is a standard which specifies how objects implemented in one component can access services provided by objects implemented in another component. The services provided by objects are grouped in *interfaces*, which are sets of semantically related functions.

New components can be added to a system at any time; i.e., a component software system is never complete. In such an evolving, decentrally constructed system it is difficult to decide whether an object is still used or whether it can be removed from the system allowing to reclaim its resources. The client of an object does not know whether other clients still have access to the same object, and a server does not know whether a client passed its reference to other clients. This lack of global knowledge can make it hard to determine whether an object is no longer referenced, and thus can be released. If an object is released prematurely, then unpredictable errors will occur. The problem is critical in component software environments, because an erroneous component of one vendor may destroy objects implemented by other vendors' components. It may even take some time until the destroyed objects

begin to visibly malfunction. For the end user, it is next to impossible to determine which vendor is to blame.

As a consequence, some form of automatic garbage collection mechanism must be provided. In contrast to closed systems, this is a necessity with component software, and not merely an option or a convenience feature.

COM uses a rather simple form of garbage collection based on reference counting. Whenever a client gets access to a COM object, it must inform the object that a new reference exists. As soon as the client releases the COM object, it must inform the object that a reference will disappear. A COM object counts its references and thus can control its own lifetime.

The problem with reference counting is that it depends on the reliability of the programmer. The situation is aggravated by additional subtle rules about who is responsible for the management of the reference counts if COM objects are passed as function arguments. Reliable software construction requires the management of reference counts to be automated. In this article we present a Direct-To-COM (DTC) compiler for the programming language Component Pascal. This compiler completely automates memory

management. Component Pascal is a modern Pascal dialect focused on component software [Pfister97].

The article is organized as follows: In Section 2 we briefly explain how COM is supported by the DTC compiler for Component Pascal, and in Section 3 we recapitulate the reference counting mechanism as specified for COM. In particular, we explain all the special rules for *in* parameters, *out* parameters and *in-out* parameters. In Section 4 we present two techniques which try to simplify COM programming: wrappers and smart pointers. The wrapper approach is provided in Microsoft's Visual J++ for its interface to COM objects, and smart pointers are provided in Release 5.0 of Microsoft's Visual C++ compiler. However, as we will see, both approaches have their disadvantages. In Section 5 we finally show how reference counting is automated in our DTC compiler for Component Pascal. We will see that this solution is superior to the other solutions, but that it puts some demands on the programming language it is embedded in.

## 2. DTC for Component Pascal

COM defines a standardized way to lay out the implementation of interfaces. A COM object provides the implementation of each method specified in the interface. The pointers to the method implementations are stored in an array, a so-called virtual function table (vtable). The interface record is a structure whose first entry is a pointer to a vtable; it may contain additional private object data. COM clients only interact with a COM object through pointers to interfaces, i.e., with pointers to pointers to vttables (see Figure 1). It is through such a pointer that the client accesses an object's implementation of the interface. Therefore, any language that can call functions via pointers can be used to program an/or use COM components.

A Direct-To-COM (DTC) compiler is a compiler that supports this binary interface standard. In our DTC compiler, COM objects are declared and implemented in a superset of the Component Pascal language. COM objects implemented in Component Pascal and compiled with the DTC compiler can be used from any other language, and COM objects implemented in any language can be used by clients compiled with the Component Pascal DTC compiler.

In Component Pascal, COM interfaces are mapped to special Component Pascal records. Interface pointers are pointers to such records. The virtual function table is the method table of the Component Pascal type descriptor. It is hidden from the programmer. A COM interface can be defined in the following way:

```

ILookUp* = POINTER TO ABSTRACT RECORD
  [{"C4910D71-BA7D-11CD-94E8-08001701A8A3"}]
  (COM.IUnknown) END;

PROCEDURE (this: ILookUp) LookupByName*(
  name: WinApi.PtrWSTR;
  OUT number: WinApi.PtrWSTR): COM.RESULT,
  NEW, ABSTRACT;

PROCEDURE (this: ILookUp) LookupByNumber*(
  number: WinApi.PtrWSTR;
  OUT name: WinApi.PtrWSTR): COM.RESULT,
  NEW, ABSTRACT;

```

Interface definitions can be derived from one another using Component Pascal subtyping. The above interface is a subtype of the interface *COM.IUnknown*. The interface identifier (IID) is specified in the declaration of the interface. The interface is declared to be abstract and cannot be instantiated. Concrete implementations of interface objects are extensions of interface records. The compiler provides standard implementations for all the methods defined in the *IUnknown* interface. For more details on programming with this DTC compiler we refer to [Gruntz97].

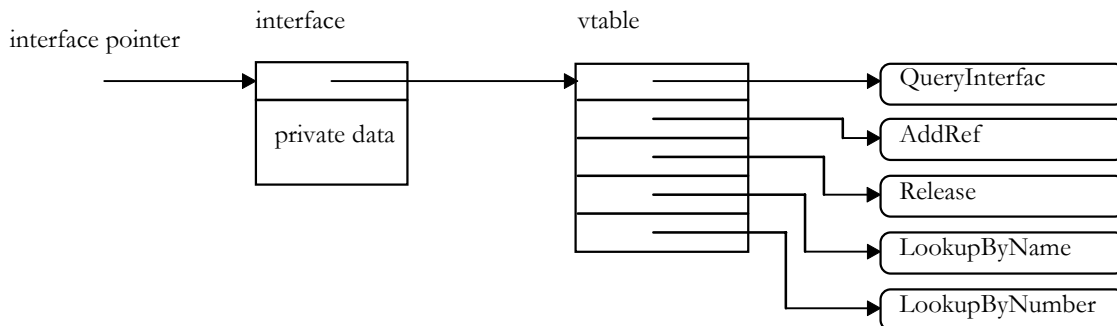


Figure 1: COM Interface layout in memory

### 3. Reference counting in COM

COM uses reference counting as a simple form of (manual) garbage collection. Reference counting is performed through two standard methods called *AddRef* and *Release*. Whenever a new reference to an interface is established, *AddRef* must be called. When the interface is no longer needed, *Release* must be called. Reference counting is done at the interface level, i.e., every interface must support the two methods *AddRef* and *Release*. They are defined in the interface *IUnknown* which is the base interface of every COM interface. Internally, the interface implementation counts the number of *AddRef* and *Release* calls, and thereby knows when it is safe to free the memory that it occupies.

Conceptually, reference counting is performed at the interface level. However, as a COM object can support several interfaces, it is free to implement one central reference count per object, instead of one reference count per interface. However, a client should never assume that an object uses the same counter for several interfaces; i.e., it should increment the reference count always through the interface which is about to be used.

Reference counting is a very simple form of garbage collection. One drawback is that it cannot release cyclic data structures. For example, two objects may reference each other, so the value of each object's reference count is one, although neither object can be accessed from anywhere else. Since cyclic data structures are very common, applications must define and follow rules to break such cycles. This is an inherent limitation

of reference counting. Although it cannot be solved generically at the level of the object model alone, a possible approach will be discussed briefly in Section 3.5.

The COM specification defines some special rules for interface parameters of methods. An interface may specify that some arguments of its methods are passed only in one direction; i.e., only from the client to the server or vice versa. This minimizes communication overhead for calls across process or machine boundaries (remote procedure calls), but it complicates the rules for the COM programmer. We will discuss these rules in Sections 3.2 and 3.3.

#### 3.1 Simple rules

The following two simple rules define how reference counts have to be managed. Interface pointer variables are variables which hold interface pointers. Such variables may be located in memory or in processor registers.

- 1) Whenever an interface pointer is stored in an interface pointer variable, *AddRef* must be called through this interface pointer.
- 2) Immediately before an interface pointer variable is cleared, overwritten, or destroyed, *Release* must be called on the interface pointer presently in the variable.

Rule 1 implies that a function that returns an interface pointer must call *AddRef* on this interface, as it stores a

```

IFactory* pFactory;
HRESULT hr = GetFactory(IID_IFactory, (void*)&pFactory);
if (SUCCEEDED(hr)) // Reference count of factory has been incremented by GetFactory
{
    IAnimal* pAnimal1;
    hr = pFactory->CreateInstance(0, IID_IAnimal, (void*)&pAnimal1); // (2)
    if (SUCCEEDED(hr)) // Reference count of pAnimal1 incremented by CreateInstance
    {
        IAnimal* pAnimal2 = pAnimal1;
        pAnimal2->AddRef(); // Increment Reference Count (1)
        pAnimal2->Sleep(); // Do something through pAnimal2

        pAnimal1->Release(); // Release Interface before assigning a new animal (1) (3)
        hr = factory->CreateInstance(0, IID_IAnimal, (void*)&pAnimal1);
        if (SUCCEEDED(hr)) // Reference count of pAnimal1 incremented by CreateInstance
        {
            pAnimal1->Eat(); // Do something with second animal
            pAnimal1->Release(); // scope in which pAnimal1 is declared will be closed soon
        }
        pAnimal2->Release(); // scope in which pAnimal2 is declared will be closed
    }
    pFactory->Release(); // scope in which pFactory is declared will be closed soon
}

```

Listing 1: Reference Counting in COM: Simple Rules

new interface pointer in a register or in memory (i.e., at a place where access is still possible). Examples of such functions are *IUnknown.QueryInterface* and *IFactory.CreateInstance*.

Rule 2 implies that *Release* must be called on old values of an interface pointer variable before assigning a new value, and on local interface pointers before leaving the scope. The example in Listing 1 demonstrates the use of these two rules.

In principle, every assignment to an interface pointer must be accompanied by a *Release* call (except for NULL or uninitialized interface pointers) and an *AddRef* call. However, in some special situations, *AddRef* and *Release* pairs can be omitted safely. The first situation is when the lifetime of one interface pointer is completely within the lifetime of another one, i.e., when a copy of an interface pointer is destroyed while the original still exists. The original then also guarantees access through the copy. Below, we will discuss two special cases of this rule in connection with *in* parameters and cyclic structures. Similarly, if the lifetimes of two pointers to the same interface overlap, i.e., if a copy is created before the original is destroyed, then the calls to *AddRef* for the copy and *Release* for the original can be omitted. A special case of this is, when a reference is moved from one variable to another one. In the example above, the two lines marked with (1) could therefore be omitted.

However, these last two special rules constitutes convenience and efficiency options, which can just be ignored, if the programmer prefers to stay on a safer side.

Unfortunately, the story is not that simple. Two more rules make programming with COM rather subtle: The *out* parameter rule and the *in* parameter rule.

### 3.2 The out parameter rule

An *out* parameter is a function parameter through which an interface pointer can be passed back to the caller. The value that the caller passes upon function call is ignored. For local (cross process) or remote (cross machine) calls this value is not even transmitted to the server. Ignoring this value means that such an interface pointer variable is overwritten without first calling *Release* on it. Therefore, the caller may pass a uninitialized interface pointer variable to an *out* parameter (see line (2) in the example above). If the passed variable contains a valid interface pointer, then *Release* must be called through this pointer before, i.e. by

the caller of the function (cf. line (3) above). The server on the other hand must assert that the returned value is a valid interface pointer or a NULL pointer (and that *AddRef* was called on it).

A result parameter is a special case of an *out* parameter. The above mentioned problems disappear, as the caller cannot pass an initial value. The server generates a new interface pointer, calls *AddRef* on it, and then stores it e.g. in a register from where the caller can fetch it. If the caller moves the value from the register to another place, rules 1 and 2 apply as usual. If the result is fetched only once, then the caller can optimize reference counting by not calling *Release* on the register reference and by not calling *AddRef* on the destination where the reference was moved to. However, interface pointers as results cannot occur in MIDL-generated and remotable interfaces. (MIDL is Microsoft's interface definition language in which COM interfaces can be specified).

### 3.3 The in parameter rule

The *in* parameter rule is the opposite of the *out* parameter rule. The value that the caller passes is transmitted to the server, but the function does not modify or return it to the caller.

*In* parameters are usually passed by value rather than by reference. This implies that a second interface pointer variable is created, e.g. on the stack. As the lifetime of this copy is nested in the lifetime of the pointer used to initialize the value, this copy is not reference counted. However, if the copy of a globally accessible interface pointer variable is passed, this interface pointer could be released during the function call through a side effect. As a consequence, the caller must guarantee that the lifetime of the passed interface pointer survives the lifetime of the function call. For globally accessible interface variables, the caller has to make a local copy and call *AddRef* and *Release* on this copy before and after the call of the function.

The function which receives access to an interface pointer variable through an *in* parameter is free to change the value of this pointer variable. However, if the variable is overwritten for the first time, no *Release* must be called, as this copy was not reference counted. If *Release* is called on an interface pointer which is not reference counted, dangling pointers will result!

### 3.4 In-out parameters

No special rule is necessary for *in-out* parameters. Conceptually, no new reference to an interface is generated when the function is called, but rather a reference to an interface pointer *variable* is passed. Therefore, neither *AddRef* nor *Release* needs to be called upon calling or returning. If the value is changed within the function, rules 1 and 2 apply as usual.

For in-process calls, *in-out* parameters are usually implemented as call-by-reference. However, for cross-process calls, pure *in* and *out* semantics has to be used, i.e., the result is copied back into the caller's address

space by the proxy object when the call returns. Both the caller and the function must not make assumptions on how *in-out* parameters are effectively implemented, and a special COM rule specifies that the function only reads the *in-out* parameter immediately after entrance in the function and only writes to the *in-out* parameter immediately before returning. If the function needs access to the *in-out* parameter during the call, it has to make a local reference counted copy of the interface pointer.

### 3.5 Cyclic structures

Uncounted reference pointers can be used to make otherwise cyclic data structures acyclic. For example, if one pointer in a ring is uncounted, the ring looks like a linear list to the reference counting mechanism. However, uncounted reference pointers should be used with extreme caution only. They are legal when the lifetime of one object includes the lifetime of a second one. Then the pointer from the second to the first object (back pointer) need not be counted. This situation occurs with COM's aggregation scheme.

## 4. Making it easier

In languages like C or C++, the programmer is responsible to call *AddRef* and *Release* when necessary. Failing to call *Release* results in memory leaks. Failing to call *AddRef* is even worse, because it results in dangling pointers. Dangling pointers are hard to track down, and they are among the most expensive programming errors. The special rules for *in* and *out* parameters complicate COM programming even further. As a consequence, several techniques have been proposed to make COM programming easier. We will present two of the most popular solutions in the context of COM programming: Wrappers and Smart Pointers.

### 4.1 Wrappers

Wrappers are used to embed COM into other programming environments, like J++. A wrapper contains pointers to one or more COM interfaces. It hides all the COM details and offers an interface of its own. The latter uses the interfacing standards of the programming environment, in which COM shall be embedded. The client calls functions of the wrapper interface only, and the wrapper forwards these calls to COM interfaces. Preferably, only wrapper classes are visible to clients. Methods that return interface pointers are mapped to wrapper functions that return references to wrappers.

Wrappers are free to add additional code before and after the call of a COM method. In particular, the wrapper can assert that the special reference counting rules are met, i.e., it can make local copies for *in* parameters or generate a new wrapper for *out*

parameters. The lifetime of a wrapped interface pointer is usually identical with the lifetime of the wrapper. In order to determine the lifetime of wrappers, a language dependent mechanism may be used, e.g. true garbage collection.

This approach is implemented in Microsoft's Visual J++. Every COM object is wrapped into a Java object, which provides its services through Java interfaces. The wrapper classes are generated automatically from a type library. Only a subset of MIDL (ODL) is supported. For example, only the two COM interface types *IUnknown* and *IDispatch* are supported as function parameter types. The Java garbage collector is responsible to remove unused wrapper objects, and the *finalize* method is calling *Release* on the wrapped interface pointers.

The price to be paid for this solution is the loss of efficiency implied by the additional indirections. Moreover, all functions provided by the wrapped COM interface have to be implemented by the wrapper, even if the implementations only consist in forwarding the calls.

### 4.2 Smart Interface Pointers

Another solution which overcomes the efficiency problem of wrappers are *smart interface pointers*. A smart interface pointer is used like an ordinary interface pointer, but the reference counting is automated to some degree.

In C++, a smart pointer is a class that contains a pointer to an interface and that overloads the *operator->* so that dereferencing is always forwarded to the embedded interface pointer (cf. [Coplien92]). Additionally, other operators are overwritten (e.g. *operator&* and *operator\**) so that the smart pointer can be used like a regular interface pointer.

In particular, the assignment operator is overwritten such that it automatically calls *Release* on the old value and *AddRef* on the one that is to be assigned. Also, the destructor of a smart pointer calls *Release*, which ensures that a local smart pointer is released automatically before the scope is left, even if the scope is left due to an exception.

Unfortunately, this solution is not as safe as the wrapper solution. It is still possible and necessary to access the embedded interface pointer directly. If a function with an *out* or an *in-out* parameter is called, the *operator&* is applied to the actual argument. For smart interface pointers, this operator returns the address of the embedded interface pointer. For *out* parameters, *Release* has to be called on the interface pointer before it is returned so that it can be overwritten, and for *in-out* parameters *Release* must not be called. As proposed in [Box96], two special routines could be offered, but the

compiler has no chance to test whether they are used properly. The same holds for the *in* parameter rule.

however, smart pointers are not as smart as wrappers are.

```

__COM_SMARTPTR_TYPEDEF(IFactory, __uuidof(IFactory)); // typedef for IFactoryPtr
__COM_SMARTPTR_TYPEDEF(IAnimal, __uuidof(IAnimal)); // typedef for IAnimalPtr

IFactoryPtr spFactory;
HRESULT hr = GetFactory(IID_IFactory, (void*)&spFactory);
if (SUCCEEDED(hr)) // Reference count of factory has been incremented by GetFactory
{
    IAnimalPtr spAnimal1;
    hr = spFactory->CreateInstance(0, IID_IAnimal, (void*)&spAnimal1);
    if (SUCCEEDED(hr)) // Reference count of spAnimal1 incremented by CreateInstance
    {
        IAnimalPtr spAnimal2 = spAnimal1; // AddrRef called automatically within operator=
        spAnimal2->Sleep(); // Automatic forwarding

        spAnimal1 = NULL; // Possibility to release embedded interface
        hr = factory->CreateInstance(0, IID_IAnimal, (void*)&spAnimal1);
        if (SUCCEEDED(hr)) // Reference count of spAnimal1 incremented by CreateInstance
        {
            spAnimal1->Eat(); // Automatic forwarding
        }
    }
}

```

Listing 2: Smart Interface Pointers

Microsoft Visual C++ provides a smart pointer solution for interface pointers in Release 5.0, with the template class `__com_ptr_t`. Listing 2 shows the code sample of Listing 1 written in Visual C++ 5.0 using smart interface pointers.

Smart pointers do not need to reimplement all the member functions of the contained COM interface and are thus more efficient than wrappers. From this point of view, wrappers are to smart pointers like COM containment is to COM aggregation. Concerning safety

## 5. Automatic garbage collection

The “Safer OLE” technology of our DTC compiler adds automatic garbage collection to COM objects. All necessary calls to the two methods *AddRef* and *Release* are generated automatically by the DTC compiler. The programmer needs not worry about reference counting at all. The compiler also automatically implements the two methods *AddRef* and *Release*. The above example written in Component Pascal is presented in Listing 3.

Calls to the reference counting methods are generated

```

VAR
    factory: IFactory; animal1, animal2: IAnimal; hr: COM.RESULT;
BEGIN
    hr := GetFactory(COM.ID(IFactory), factory);
    IF hr >= 0 THEN
        hr := factory.CreateInstance(NIL, COM.ID(animal1), animal1);
        IF hr >= 0 THEN
            animal2 := animal1;
            animal2.Sleep();
            hr := factory.CreateInstance(NIL, COM.ID(animal1), animal1);
            IF hr >= 0 THEN
                animal1.Eat()
            END
        END
    END
END
END

```

Listing 3: Component Pascal

for assignments to interface pointer variables. The garbage collector recognizes COM interface implementations and does not remove such objects with a reference count greater than zero. If the garbage collector finds a COM interface object with reference count zero, then all interface pointers stored in this object are released before the object is collected. The same holds for local variables if the scope is left and for global variables that are stored in a component to be unloaded.

### 5.1 Interface function parameters

In order to automate the reference counting calls for function parameters, the compiler must know whether a parameter is an *in* parameter, an *out* parameter, or an *in-out* parameter. In Component Pascal this can be specified explicitly. Below you see the specification of an interface of a garage where cars can be bought (*out*), checked in for service (*in*), and be repaired (*in-out*).

```

TYPE
  IGarage = POINTER TO RECORD
    [{"44660001-0fa3-11cf-adf0-444553540000"}]
    (COM.IUnknown)
    PROCEDURE (g: IGarage) BuyCar (OUT car: ICar):
      COM.RESULT, NEW, ABSTRACT; (* out *)
    PROCEDURE (g: IGarage) CheckCar (car: ICar):
      COM.RESULT, NEW, ABSTRACT; (* in *)
    PROCEDURE (g: IGarage) RepairCar (VAR car: ICar):
      COM.RESULT, NEW, ABSTRACT; (* in-out *)
  END;

```

For functions with interface pointer parameters, the compiler automatically generates the necessary code for reference counting. This support is provided both on the client side and on the server side, that is, the compiler adds necessary code to the call of functions provided by COM interfaces, and also to interface function implemented in Component Pascal.

For *out* parameters, the compiler creates code on the client side to release the actual argument before it is passed to the interface pointer variable. On the server side, all *out* parameters are initialized with the NULL pointer.

For (value) *in* parameters, the compiler guarantees the liveness condition on the client side. If the variable being passed is not local, the compiler generates a local copy and calls *AddRef* on this copy before and *Release* after the call. On the server side, the compiler asserts that no *Release* is called on this parameter, in particular not upon termination of the function. However, if an *in* parameter gets assigned inside a function, the compiler generates code to increment its reference count upon function call and treats it as a regular interface pointer.

### 5.2 Aggregation

Aggregation requires special treatment. When reusing a COM object through aggregation, two objects are involved: the outer one and the inner one. The outer object has a reference to the inner *IUnknown* interface,

in order to use interfaces that are implemented by the inner object. The inner object has a reference to the outer *IUnknown* interface, in order to forward *QueryInterface* and reference counting calls. As a consequence we would have a cycle which is reference counted, i.e., even if the cycle is no longer used, all reference counted objects in this cycle would have the reference count 1 and thus the whole cycle could not be removed. Therefore, an easier solution is proposed in the COM style guide: The reference from the inner object to the outer object is not reference counted. This is safe, as the outer object cannot disappear as long as the inner object is needed.

In our DTC compiler such uncounted references can be generated, but they are hidden from the programmer. They are used in the hidden implementations of *AddRef* and *Release* and in the default implementation of *QueryInterface* to forward the requests. If one of these three functions is called on an interface which has a hidden uncounted reference to another object, then the call is always forwarded to the referenced object. Note that such a construct is not only useful for aggregation but also to implement COM objects which implement several interfaces.

Another problem appears if an object which reuses another object through aggregation makes a local copy of an interface pointer which it obtained from the reused object, e.g. to maintain a cache. If the outer object calls *AddRef* on such a pointer, it increments its own counter due to the forwarding of the *IUnknown* method calls. As a consequence, this object cannot be removed by the garbage collector. In C++, uncounted interface pointers are used in this situation. With the DTC compiler, the outer object should not keep cached interface pointers, or alternatively it has to clear the cache from time to time.

### 5.3 Interface result parameters

The DTC compiler also handles calls of functions which return interface pointers (although such functions cannot be called across process boundaries, as MIDL does not support such signatures for interface functions). Consider a function *GetID* which returns a pointer to the *IUnknown* interface for a given interface.

```

PROCEDURE GetID(int: COM.IUnknown): COM.IUnknown;
  VAR iunk: COM.IUnknown; hr: COM.RESULT;
BEGIN
  res := int.QueryInterface(COM.ID(COM.IUnknown), iunk);
  RETURN iunk
END GetIUnknown;

```

This function returns a pointer which identifies a particular COM object, and thus can be used to compare two COM objects. With our DTC compiler such a comparison can be written directly as

```
IF GetID(car1) = GetID(car2) THEN ....
```

When this statement is executed, then two new interface pointers are generated through which *AddRef* is called. The interface pointers are stored on the stack. As they are no longer used after the comparison, the compiler generates the code to release the two interface pointers on the stack after the comparison. A similar situation which is also handled automatically occurs if e.g. the expression *GetID(car1)* is supplied for an interface value (*in*) parameter.

#### 5.4 Interface inspector

The integrated development environment provides an interface inspector. Since the run-time system has full control over all allocated interface objects, they can be displayed together with their actual reference counts (see Figure 2). Objects with reference count zero are With the help of this interface inspector we were able to detect a programming error in another (third-party) application. We re-implemented the object which this other application was managing with our DTC compiler, and thus could prove that the client of our objects was not releasing them properly.

The DTC Component Pascal compiler is fully integrated in the BlackBox Component Builder. For example, a special interface browser allows developers to see the definition of COM interfaces and to view the layout of their method tables. For more information on the DTC Component Pascal compiler and the BlackBox Component Builder we refer to [Ominc97].

## 6. Summary

We have presented several techniques to support COM programmers when handling reference counts of COM objects. We described how our DTC compiler handles reference counting automatically. This automatic garbage collection mechanism just generates code that would have to be written manually in any case. Thus it does not require any extra run time and is fully compatible with other components, created without our tools.

The special parameter passing rules require the compiler to be informed about the parameter modes (*in*, *out*, *in-out*). Therefore, the programming language of a DTC compiler which automates memory management for interface pointers must support such parameter passing modes (as e.g. Component Pascal or Ada).

Microsoft is considering to build true garbage collection into COM, i.e., to let the system handle memory management, rather than the language or the programmer. No further details are known at the time of this writing, but it proves that the fundamental importance of automated garbage collection in component software environments is increasingly being acknowledged. Until then, our DTC compiler, which is

removed upon the next call of the garbage collector, if they are not referenced through a regular Component Pascal pointer (which are not interface pointers).

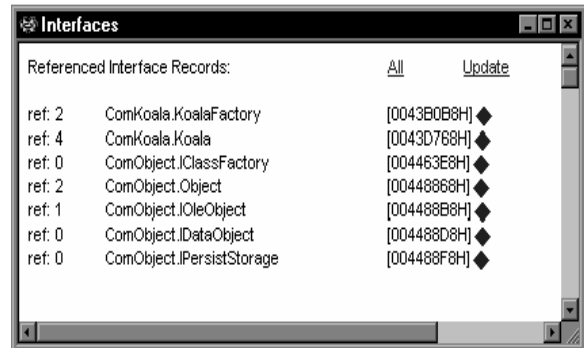


Figure 2: COM Interface Inspector

available now, presents a good answer to COM's reference counting clinches.

#### Acknowledgements:

Thanks to Matthias Hausner, Cuno Pfister and Wolfgang Weck for their many useful comments and corrections on earlier versions of this paper.

#### References:

- [Box95] Don Box,  
*Interface Pointers Considered Harmful*,  
C++ Report, September 1995,  
<http://www.develop.com/dbox/cxx/InterfacePtr.htm>.
- [Box96] Don Box,  
*COM Smart Pointers Considered Harmful*,  
C++ Report, February 1996,  
<http://www.develop.com/dbox/cxx/SmartPtr.htm>.
- [Brockschmidt95] Kraig Brockschmidt,  
*Inside OLE*,  
2<sup>nd</sup> Edition, Microsoft Press, 1995.
- [Coplien92] James O. Coplien,  
*Advanced C++ Programming Styles and Idioms*,  
Addison-Wesley, 1992, ISBN 0-201-54855-0.
- [Gruntz97] Dominik Gruntz,  
*Implementing COM Objects with the Direct-To-COM Compiler*,  
The Oberon Tribune, Vol 2, No 1, p. 7-9, 1997,  
<http://www.oberon.ch/services/odf/tribune/trib3/ComExample.html>.
- [Ominc97] Oberon microsystems Inc.,  
*Direct-To-COM Compiler*,  
<http://www.oberon.ch/prod/>.
- [Pfister97] Cuno Pfister,  
*The Language Component Pascal*,  
[http://www.oberon.ch/docu/component\\_pascal.html](http://www.oberon.ch/docu/component_pascal.html).
- [Rogerson97] Dale E. Rogerson,  
*Inside COM*,  
Microsoft Press, 1997, ISBN 1-57231-239-8.