The following paper was originally published in the
Proceedings of the USENIX 1996
Conference on Object-Oriented Technologies
Toronto, Ontario, Canada, June 1996.

Interlanguage Object Sharing with SOM

Jennifer Hamilton
IBM Toronto Laboratory

Interlanguage Object Sharing with SOM

Jennifer Hamilton
C++ Compiler Development
IBM Toronto Laboratory
jenniferh@vnet.ibm.com

Abstract: Object-oriented programming languages may encourage reuse at the source code level, but they inhibit reuse at the binary object level. Differences in object representation make it much more difficult to share objects, even across different implementations of the C++ language, than to share libraries between different procedural languages such as C and Fortran. IBM has addressed this problem through the System Object Model (SOM). The purpose of this paper is to provide a brief description of the SOM and the mapping from SOM to the object models of several languages: C++, Smalltalk, OO COBOL, and to discuss how binary object interoperability can be achieved through SOM.

1. Overview

The IBM System Object Model (SOM) was designed with three major goals: to enable release-to-release binary compatibility (RRBC) of classes, to provide a state-of-the-art object model, and to facilitate inter-languages sharing of objects [10]. While there has been examination of the success of the first two goals, there has been little investigation of SOM's ability to support mixed-language applications and to enable binary object interoperability. Partly this is due to the fact that, until recently, SOM support was only available for the C and C++ languages, which have very similar language models, so interoperability between these languages cannot be considered generally conclusive. Recently, however, SOM support has been introduced for two additional languages: Smalltalk and OO COBOL.

There are two questions to be answered through this paper. The first is whether binary object interoperability is even possible through SOM among such diverse languages as C++, OO COBOL, and Smalltalk. The second, and more interesting, is to examine the feasibility of using DirectToSOM C++ classes (SOM support where classes are defined using the full C++ language syntax) from other languages. This is an important issue because it would provide additional markets for C++ class library vendors who ported their classes to DirectToSOM C++, thereby increasing the set of class libraries available for use from other languages.

2. Introduction

Object-oriented programming languages provide many well-established advantages over conventional procedural programming languages, in particular through support for encapsulation, which groups data with associated methods. However, this grouping also introduces some problems, specifically in the area of release-to-release binary compatibility and interlanguage object sharing. The C++ language, arguably the most commonly used object-oriented programming language, suffers in particular from these problems.

With procedural languages, new versions of library routines can be introduced without impacting existing code, provided that the procedure signatures are kept compatible and new procedure names don't collide with existing client names. While keeping signatures compatible and avoiding name collisions can sometimes be difficult, it is a relatively simple problem compared to that of keeping class definitions compatible in languages such as C++. The problem for C++ is that it is a static language with a large amount of information about the class, such as its instance size, the order and location of methods, and the offset to parent class data, compiled into client code. Thus adding a new data member to a class, even a completely private member, in most cases requires recompilation of client code, including subclasses. In some cases, binary compatibility can be achieved by carefully managing class changes, but migrating a method up the class hierarchy or inserting a new class in the hierarchy always requires recompilation of client code. Languages such as Smalltalk, where class information is managed dynamically rather than statically, do not have this problem.

Object-oriented programming languages also impede the sharing of code between languages. It is relatively easy to call a C library routine from Fortran, or vice-versa, but very difficult, if not impossible, to share objects between languages such as Smalltalk and C++. This is because each language introduces a specific, internal structure for representing object data and associated methods. There is no standard object representation, such as operating system linkage conventions for procedural languages, to enable the sharing of objects across different languages. Even within a programming language, object sharing is not readily achievable. This is a particular problem for C++: there is no standard object representation defined for the language, so each compiler implementer must choose a layout. Unless the layout is identical between two compiler vendors, objects cannot be shared between these implementations.

Object-oriented programming is intended to promote code reuse and allow changes to be made to class implementations without affecting client code. This source level solution leads to a new set of problems with release-to-release binary compatibility and interlanguage object-sharing. As there is much work being done in the area of class libraries and frameworks, it is particularly important to solve this binary object problem so that class library providers can supply updated versions of their classes without forcing recompilation of existing client code. Further, class libraries should be usable from different languages, or at the very least different language implementations, without requiring multiple versions of the library for each target language or implementation.

3. SOM

The *System Object Model* (SOM) was designed to address the two problems introduced by object-oriented programming languages: release-to-release binary compatibility and interlanguage object sharing. SOM provides separation of interface and implementation through a language-independent object model, allowing the class implementation and client programs to be written in different languages. SOM allows a new version of a class to be supplied without requiring recompilation of any unmodified client code. In general, making a change to a SOM class that does not require a source code change in a client, such as adding new methods, instance variables, or even additional base classes, does not require recompilation of that client.

SOM class interfaces are defined using the OMG CORBA (see [2]) standard language called the *Interface Definition Language* (IDL), which is language-independent although loosely based on the C++ language. As an example, the following shows the IDL definition for the SOM class `Hello` with a single method `sayHello`.

```
#include <somobj.idl>

interface Hello : SOMObject
{
    void sayHello();
};
```

The SOM IDL compiler generates *language bindings* for the target client and implementation language corresponding to an IDL class definition. Bindings are language-specific macros and procedures that allow a programmer to interact with SOM through simplified syntax that is natural for the particular language. For example, the C++ bindings allow SOM objects to be manipulated through C++ pointers to objects. Currently, the SOM IDL compiler generates bindings for C and C++.

The SOM run time controls the layout and direct manipulation of class instances. All manipulation of SOM objects is performed through standard procedure calls to the run time. The language bindings provide mechanisms to map the native language syntax to SOM run-time calls. As an alternative to defining SOM classes using IDL, several compilers provide *DirectToSOM* (DTS) support, which allows a class to be defined and manipulated completely using the given language, without ever generating IDL. For example, the IBM C++ compilers for OS/2, Windows, AIX, and MVS allow you to define classes in C++, which they then map to SOM classes implicitly.

Figure 1 shows the relationship between the SOM class description mechanisms and the run-time model. Classes are described either using IDL or through native language syntax with a DTS compiler. If using IDL, the SOM compiler generates language bindings for the client and the implementation, and the corresponding language compilers are used to create binaries using the language bindings. No special compiler support is required to process the language bindings. A DTS compiler generates the client and implementation binaries directly. Note that a class client or implementation could be written using a language for which no language binding or DTS support is available ("other client" in Figure 1). SOM objects can be accessed from any language that supports external procedure calls and procedure pointers and that can map IDL types onto the native language types. The client and implementation interact through the SOM run time support. The arrow between the client and the SOM run time is single-ended, representing a one-way relationship, while the arrow from the SOM run time to the class implementation is double-ended because the SOM run time uses the implementation (for allocation, initialization, and destruction of class instances, among other things).

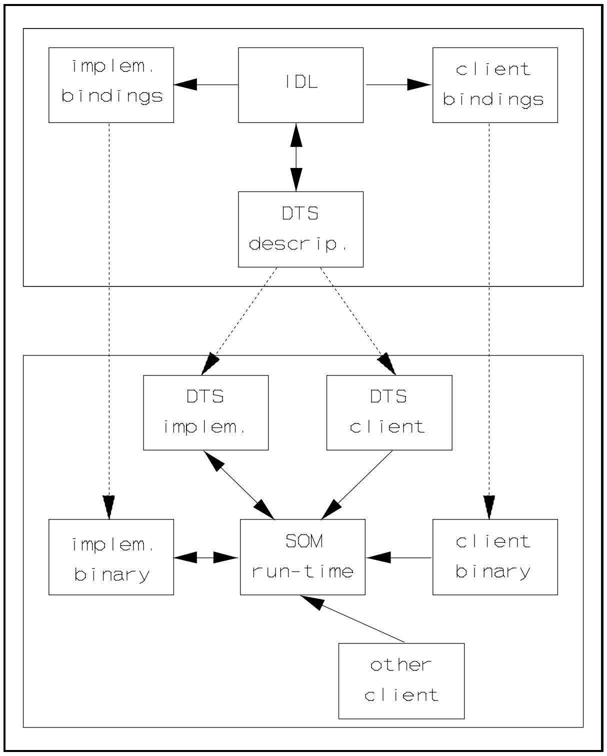

Figure 1 SOM Object Model

3.1 SOM Objects

SOM objects are run-time entities that support a specific interface and have an associated state and implementation. The implementation is only accessible through the SOM object. SOM supports a model similar to that of Smalltalk, in that classes are not purely syntactic entities, as in C++, but are themselves SOM objects. SOM class objects are created at runtime as required by the client, and are used to create and manipulate instances. Class objects support a variety of methods for creating and querying objects, such as determining the size of class instances, whether a method is supported by a given class, and whether a given instance object is a member of that class. A class object is an instance of a special kind of class, called a *metaclass*.

Methods may be invoked on a SOM object in several ways: *offset resolution*, *name-lookup resolution*, and *dispatch-function resolution*. With offset method resolution, the client code invokes the method through a *method token* found at a specific offset in a run-time table. The method token offset is known at compile-time. Name-lookup resolution, by contrast, uses the name of the method to search for the method token. Dispatch-function resolution allows the receiving object to control how the method resolution is performed. Offset resolution is the most efficient means of invoking a method, because the method token is available statically, but the client code is dependent upon the location of that method token not changing. The fixed ordering of the method token table is established by the *release order* for the class.

Every class has a release order, which is simply an ordered list specifying all methods introduced by that class. A client using offset method resolution determines the offset for a method token at compile-time according to that method's location in the release order (which is handled implicitly by language bindings). If a new method is added to the class, at the end of the release order list, it shows up at the end of the method token table, and thus will not impact existing client code. The release order list is the only dependency that a client has upon a corresponding class implementation.

For static clients using offset method resolution to invoke class methods, methods in the release order cannot be removed or reordered without breaking RRBC. New methods can be added only to the end of the release order. Dynamic clients that use name lookup or dispatch-function resolution have no dependencies upon the release order list, and will not be affected if the list is reordered. However, deleting a method from a class could result in a run-time error if that method were later invoked by a dynamic client, becuase that method would not be found.

3.2 Interface Repository

The SOM compiler can optionally create a database, called the SOM *Interface Repository* (IR), which contains class information as supplied by the IDL description. The database can be queried through SOM APIs so that a at run-time a program can access any information available about a class interface. The interface repository content and programming interface conform to those defined by OMG's CORBA Interface Repository. Among other things, the IR provides another mechanism for programming languages to support interaction with SOM. Specifically, Smalltalk language bindings are generated from the IR by the Smalltalk SOM, while OO COBOL uses the interface repository directly, instead of language bindings, to access information about existing SOM class descriptions.

Figure 2 summarizes how the various programming languages that currently provide SOM support access and create class descriptions through languages bindings, IDL, and the interface repository. The next few sections cover the SOM support for C++, OO COBOL, and IBM Smalltalk.

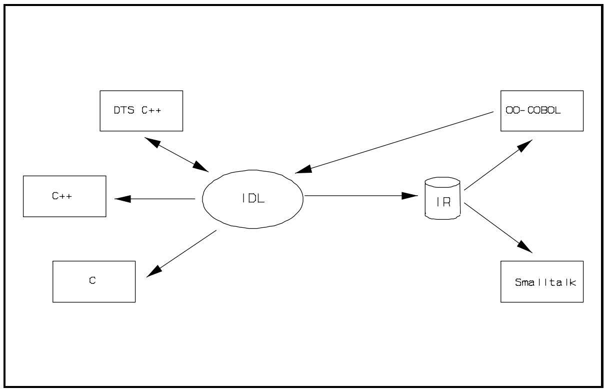

Figure 2 Language Access to SOM Class Descriptions

3.3 DirectToSOM Support

Instead of describing a SOM class using CORBA IDL, DirectToSOM (DTS) support for a programming language allows the class to be described completely in the native implementation language. The compiler generates the appropriate SOM calls and symbols for the class implementation and clients. IDL can be generated from the native language class description if required, or all SOM interactions can be done completely within the native programming language. A subcategory of DirectToSOM, which we call DirectFromSOM, gives client-only capability using native language syntax.

DirectToSOM support is currently supported by two programming languages: C++ and OO COBOL, while DirectFromSOM is supported through IBM Smalltalk. As an example, the code segment below shows a definition for a simple DirectToSOM C++ class. A C++ class is made into a DTS C++ class by inheriting from the class `SOMObject`, which is defined in the header file `<som.hh>`. The access specifiers private, protected, and public are supported for SOM classes and enforced following the C++ rules, as are constructors and destructors and most other C++ constructs. The DTS class definition can be used directly by both class client and implementation programs; no IDL description is required.

```
#include <som.hh>

class Hello : SOMObject {
  public:
    void sayHello();
};
```

4. Using SOM with C++

C++ programmers can define SOM classes in one of two ways: either through the C++ language bindings generated from an IDL description, or directly in C++ using a DirectToSOM C++ compiler. The capability to generate C++ bindings from an IDL description allows SOM objects to be created and manipulated with any C++ compiler, gaining the advantages of the RRBC support provided by SOM. In addition, those objects can be shared across different C++ implementations or even with different languages such as Smalltalk. However, in using the C++ bindings, you are limited to a subset of the C++ language, making migration of existing C++ applications more difficult, and you must use two languages (IDL and C++) to define and manipulate objects.

DirectToSOM (DTS) C++ compilers support and enforce both the C++ and the SOM object models, allowing C++ programmers to take advantage of SOM through C++ language syntax and semantics. This makes the use of SOM reasonably transparent and efficient. Instead of first describing SOM classes in IDL, the DTS C++ compiler translates C++ syntax to SOM. You can then have the compiler generate IDL from your C++ declaration, or you may find that you don't need to deal with IDL at all and can work exclusively in DTS C++. And, because you write C++ directly, you can use C++ features in your SOM classes that aren't available through the language bindings, features like templates, operators, constructors with parameters, default parameters, static members, public instance data, and more. The DirectToSOM support is of particular interest in this paper, as it allows existing classes to be migrated to SOM within the confines of the C++ language. Further details about the DirectToSOM C++ support can be found in [4], [5], [6] and [7].

A C++ class is made into a DTS C++ class by inheriting from the class `SOMObject`, which is defined in the header file <som.hh>. You can do this explicitly, as shown above, or implicitly, through compiler switches or pragmas that insert `SOMObject` as a base class. The access specifiers `private`, `protected`, and `public` are supported for SOM classes and enforced following the C++ rules, as are constructors and destructors and most other C++ constructs. You can create SOM objects statically or dynamically, as simple objects, arrays, or as embedded members of other classes, or anywhere else that the declaration of a C++ object is valid. Most of the C++ rules and syntax apply to DTS classes and objects, with some restrictions. Because the size of a SOM object is not known until run time, compile-time constant expressions such as `sizeof` are treated as run-time constant expressions. Such operators can still be used with SOM objects, but not in contexts that require compile-time evaluation.

A major inhibitor to RRBC with C++ is the fact that so much information about an object is statically compiled into client code, in particular the location of instance data and virtual function pointers. Data layout and method calling for a DTS C++ class are done using the SOM API, instead of the native C++ API. When you run a program defining a DTS C++ class, the compiler creates the corresponding SOM class object at run time and uses it to create and manipulate the object. As a result, unlike a standard C++ object, much of the information about a SOM object and its class, such as the instance size, is not determined until run time, when the class object is created. This enables class evolution without forcing recompilation of client applications.

C++ instance data members in a DTS class are regrouped into contiguous chunks according to access, in the order of declaration within the class. This regrouping gives efficient access to data members from client code, while enabling RRBC. The location of each chunk is determined at run time. If the declaration order of public and protected data within a class is not changed, and new members are added after any pre-existing members of the same access, this scheme allows new data members to be added without requiring recompilation of any code outside the class (except for friends).

A DTS class also has a default release order. It contains, in the order of declaration, all member functions and static data members introduced by the class, including those with private and protected access. Using the default, you must add any new member functions or static data members at the end of the class. Instead of relying on declaration order, you can instead use the a pragma to specify the release order, in which case you can add new release order elements anywhere in the class, but you must add their names to the end of the list.

For DTS classes, instance data and the release order list are accessed through the SOM run time when manipulating SOM objects, rather than through the statically-defined compiler constructs used by standard C++. This approach provides for both RRBC and an implementation-independent object model. As long as the order of list elements does not change and new elements are added to the end of the list, you can add new data members and member functions without forcing recompilation of client code. In the same way, you can migrate a member function up the class hierarchy. This model solves the fragile base class problem, allowing changes to be made to a base classes without forcing recompilation of derived classes. Further details on the support and restrictions of the model can be found in [4] and [5].

5. Smalltalk

The Smalltalk support for SOM, Smalltalk SOMSupport, is currently available as client-only DirectFromSOM support through the IBM VisualAge for Smalltalk product. The SOM compiler does not create language bindings for Smalltalk from IDL, rather the Smalltalk SOMsupport uses the SOM Interface Repository to create native Smalltalk *wrapper classes* for specific SOM objects. Wrapper classes are generated explicitly through a framework of classes provided with the SOMsupport. The wrapper classes can only be used to create and manipulate SOM objects as a client, there is no support for implementing SOM objects from within Smalltalk, either explicitly, or implicitly by inheriting from one of the wrapper classes. Further details about the constructor and the Smalltalk SOMSupport can be found in [9].

The generated wrapper classes have the same name as the SOM class, prefixed with the string SOM. In order to conform to Smalltalk naming conventions, underscores in class or method names are removed, and capitalization is changed (for example, the first letter of the class name or the letter after an underscore is capitalized). Once the wrapper classes are generated, they become part of the set of classes available to the application. These wrapper classes provide methods that can be invoked using standard Smalltalk syntax, but they map to calls to the SOM API. When an instance of a wrapper class is created, a corresponding SOM class instance is also created. Methods invoked on the wrapper instance result in the invocation of that method in the corresponding SOM instance. As an example of using a SOM class from within Smalltalk, the following shows a Smalltalk code sequence that creates a SOM object of class `Hello`, invokes the method `sayHello`, and deletes the object.

```
| obj |

obj := SOMHello new.
obj sayHello.
obj somFree.
```

A major consideration in mapping from Smalltalk to SOM is memory management. When a Smalltalk object is no longer in use, it is automatically freed by the Smalltalk garbage collector. SOM objects exist outside the Smalltalk memory space, and thus must be destroyed explicitly through the `SOMObject` method `somFree`. Because the absolute memory address of a Smalltalk object can change as the memory manager reallocates storage, Smalltalk cannot pass the actual address of an object as a parameter to a SOM method. Instead, it copies the object to SOM memory and passes the address of the SOM memory copy. However, this copy exists only for the duration of the method call, so SOM implementation methods cannot reliably store this address and use it later. Thus, if the object needs to be referenced in the future from within the implementation, a deep copy of it must be made, rather than simply storing the address.

6. OO COBOL

The proposed ANSI OO COBOL definition [1] extends COBOL-85 with support for object-oriented programming. There are no COBOL bindings for SOM, as there are for C++, which would allow any OO COBOL compiler to access SOM. Instead, the IBM COBOL compilers provide DirectToSOM support by using SOM as the native object model for implementing the OO extensions. Native language syntax is used to define SOM objects, as with DirectToSOM C++, although the use of SOM is much more explicit with OO COBOL. Therefore, for the purposes of this discussion, language interoperability through SOM, the COBOL discussion is restricted to the support provided by IBM OO COBOL.

IBM OO COBOL supports a subset of the proposed ANSI OO COBOL definition. The spirit of the IBM OO COBOL definition is to provide essential OO building blocks through native support, but to take advantage of the existing support in the SOM API wherever possible. To that end, the language description for these extensions is fairly brief. Native language support, based on the proposed ANSI definition, is provided for describing classes and their methods, defining object variables, and method invocation. The object model itself, however, and most other object support, is implemented using SOM; for example, objects are created and destroyed by invoking SOM methods. The following is a brief overview of the native language support. Further details can be found in [8].

6.1 Class Definition

New native language syntax is provided to define a class. All classes must directly or indirectly inherit from SOMObject, which implies that a class must always have at least one direct parent. Following the SOM model, a class is also an object with a metaclass of `SOMClass`, although the class definition may designate a different class as the metaclass. All metaclasses must derive from `SOMClass`. Multiple inheritance is supported, the detailed semantics of inheritance being defined by SOM.

Instance data introduced in a class is accessible only by the methods introduced by that class and is private otherwise. A class cannot access the instance data of a parent class or metaclass. There is no provision for defining class data that is shared across all classes through the class definition itself, although this can be done through a metaclass.

IBM OO COBOL supports both the use and implementation of SOM objects. The compiler uses the SOM Interface Repository to extract information about referenced classes to perform compile-time static type-checking, such as ensuring that a specified object type supports a given method and that the parameters and arguments are compatible. The compiler also supports generation of IDL definitions for SOM classes defined in the COBOL program. Thus a class implemented in C++, for example, can be used by an OO COBOL program, as an OO COBOL class implementation can be used by a Smalltalk program.

6.2 Methods

A class inherits all methods defined by its parent classes. There is no mechanism for hiding a method introduced by a parent class. A class may introduce new methods and override one introduced by a parent class to provide a different implementation. Name overloading by method signature is not supported; all methods introduced by a class must be uniquely named within that class, regardless of case. When a parent method is overridden, the signatures of the overridden and overriding method must be compatible, which, depending upon the parameter type, means that they must be of the same class or one must be a parent class of the other. If the same method name is introduced by two different parents classes, the signatures for those methods must be compatible and that of the leftmost class in the hierarchy is used. This behavior can be modified by overriding the method in the new class and explicitly invoking the desired method by class name. Method binding is performed dynamically using name-lookup resolution.

There is no native support to automatically define CORBA attributes, but this can easily be achieved by introducing the appropriate _get and _set methods.

6.3 Initialization

No native support or syntax is provided for the creation and destruction of objects. Instead, objects are created by invoking the `SOMClass` method `somNew` against a class object and destroyed through the `SOMObject` method `somFree`. Objects can be automatically initialized and deinitialized by overriding the `SOMObject` methods `somInit` and `somUninit`.

To illustrate the OO COBOL support, the following shows a simple example of a class definition, `Hello`, and a client written in OO COBOL. The two programs are compiled separately and then statically linked together.

Definition of OO COBOL Class `Hello`

```
Identification Division.
Class-id. Hello inherits SOMObject.
Environment Division.
Configuration section.
Repository.
    Class SOMObject is "SOMObject"
    Class Hello is "Hello".
Procedure Division.

Identification Division.
Method-id. sayHello.
Procedure Division.
    Display "Hello from COBOL".
End method sayHello.

End class Hello.
```

Cleint of OO COBOL Class `Hello`

```
Identification Division.
Program-id. Client.
Environment Division.
Configuration section.
Repository.
    Class Hello is "Hello".
Data Division.
Working-storage section.
01 obj usage object reference Hello.
Procedure Division.
    Display "Calling somNew".
    Invoke Hello "somNew" returning obj
    Display "Calling sayHello".
    Invoke obj "sayHello".
    Display "Calling somFree".
    Invoke obj "somFree"
    Goback.
End program Client.
```

7. Defining Language-Independent DTS C++ Classes

This section describes the basic considerations for designing DTS C++ classes that can be used from other SOM-enabled languages, and serves partially to explain the coding practices used in the examples that follow.

7.1 Class and Member Names

One of the major considerations with interlanguage sharing of DTS C++ classes is name mangling. Because IDL is case-insensitive and does not support name overloading within a class, C++ member and class names are mangled to provide unique IDL names. In order to accommodate this, C++ names are mangled using a SOM mangling scheme that loosely follows the mangling scheme used by most C++ compilers. In addition, uppercase characters are converted to lowercase by prefixing them with a lowercase z. z_ is used to mean a real lowercase z. So `somSayHelloZz` becomes `somzsayzhellozzz_`.

Within DTS C++, name mangling does not pose a problem; however, the mangled names tend to be fairly long and unreadable, making them unsuitable for use by other languages. There are several pragmas that can be used to affect DTS C++ name mangling for SOM: `SOMNoMangling`, `SOMMethodName`, `SOMDataName`, and `SOMClassName`.

`SOMNoMangling` prevents the name mangling of class member names and can be turned on for a specific class or for a range of classes in the compilation unit. If the class has overloaded member functions, this causes collisions in the generated IDL, in which case the `SOMMethodName` pragma is also required to give specific names to overloaded members. `SOMDataName` can be used to give specific names to class data members. `SOMNoMangling` does not affect class names, which are at the very least mangled to be case-insensitive by translating uper case letters to the lowercase equivalent preceded by lowercase z. Template class names are mangled further to incorporate type information. Therefore, if the class name contains uppercase letters or is a template class, the

SOMClassName pragma should also be used to ensure that the class name is not mangled.

Note that IDL matches overloaded methods by name only, so if a name is not mangled initially, but is mangled later on, this will break binary compatibility because it is equivalent to changing the method name or to removing the method with the old name and adding a method with the new name. Methods cannot be deleted from a SOM class without potentially breaking binary compatibility. The recommended approach if interlanguage sharing is likely to be required is to always use SOMNoMangling and SOMClassName, and to use SOMMethodName when necessary to handle overloaded member names or special names such as operators.

Mangled or not, names should contain only alphabetic characters or digits, and should begin with an alphabetic character. Underscores, while valid for IDL names, may cause problems in languages such as Smalltalk, that remove them from names.

7.2 Data Members

Most other languages do not have the ability to directly access public or protected data members or static data members. The simplest way to allow other languages to access public data members is by making them into CORBA attributes. This can be done using the SOMAttribute pragma. The DTS C++ compiler implicitly generates _get_*member* and _set_*member* methods that return and set the member value respectively. Note that by default the backing data becomes private and all C++ access to the data members outside the class is through the attribute functions. For performance reasons, the backing data can be made public for direct access from DTS C++.

For public static data members, which are shared across all class instances, the recommended approach is to define a metaclass for the given class and make the static data member an attribute of the metaclass.

7.3 Constructors

SOM requires that a class define a default constructor with no arguments. This constructor is mapped by the DirectToSOM C++ compiler to an override of the SOMObject method somDefaultInit. C++ copy constructors override one of four SOMObject copy constructor methods depending upon the method signature (somxxCopyInit, where xx is determined by the source object being const or volatile). Any other C++ constructors will have mangled names, becuase they are not mapped to any SOMObject methods, and should explicitly be given SOM names via the SOMMethodName pragma.

Through both OO COBOL and Smalltalk, somNew can be invoked to create objects. somNew implicitly calls somDefaultInit as part of object creation, which in turn calls the C++ no-argument default constructor. Object creation can also be performed in two steps by first creating an uninitialized object through invoking the SOMClass methods SOMNewNoInit or SOMRenewNoInit against a given class object, and then invoking somDefaultInit against the object.

This two-step mechanism can be used to call copy constructors, such as somDefaultConstCopyInit or other constructors, from OO COBOL or Smalltalk, which don't have language mechanisms to implicitly call these copy constructors. All SOM constructor methods accept three parameters: the target object, an environment, and an initialization control vector. The initialization control vector is used to prevent a class constructor from being called more than once when a class appears multiple times in the inheritance tree, and it should always be supplied as a null pointer.

7.4 Assignment

If an operator= method is not defined for the class, the compiler supplies overrides of the four SOMobject assignment methods (somDefaultxxAssign, where xx is determined by the source object being const or volatile). One of these methods is called when an assignment operator is encountered. If an operator= method is supplied, then it is called when one of the SOM assignment methods is invoked. The SOM assignment methods accept a first parameter that is an assignment control vector, followed by the source object for the assignment. As with the initialization control vector, the assignment control parameter is always passed as a null pointer and is used to prevent a base from being assigned more than once when it appears multiple times in the hierarchy. This support is not available for operator=. So, a SOM assignment method should be defined in preference to an operator= method.

8. Examples

This section provides examples of sharing code with DirectToSOM C++ (IBM VisualAge C++ for OS/2 Version 3.0.), OO COBOL, (IBM VisualAge for COBOL for OS/2 Version 1.1) and Smalltalk (IBM VisualAge for Smalltalk for OS/2 Version 2.0). The first example is a simple DTS C++ class `Hello`, defined below. Note the use of the `SOMNoMangling` and `SOMClassName` pragmas to control IDL name generation. The `SOMIDLPass` pragma is used to add information to the IDL file that cannot be expressed in C++. In this case, it is used to indicate the name of the DLL that contains the current class. The DLL name is used by the SOM API when a class is dynamically loaded at runtime rather than statically bound to the application. SOM classes should be designed for both situations. The corresponding IDL generated by the DirectToSOM C++ compiler is shown following the class definition.

The C++ client program shows how this class is used in C++ using standard C++ syntax, followed by examples of using this class from OO COBOL and Smalltalk respectively. In OO COBOL, the `somNew` method is invoked against the class object `Hello`, which returns an instance of the class `Hello`. Then the `sayHello` message is sent to the object, and finally the object is deleted via `somFree`. The same approach is followed for Smalltalk. Note that the SOM object must be explicitly freed; the Smalltalk garbage collector does not handle SOM objects. The class implementation is shown at end, written using standard C++ syntax.

Definition of DTS C++ Class `Hello`

```
#include <som.hh>

#pragma SOMNOMangling(on)

class Hello : public SOMObject {
    #pragma SOMClassName(*, "Hello")
    #pragma SOMIDLPass(*, "Implementation-End", "dllname = \"hello.dll\";")
  public:
    void sayHello();
};
```

Generated IDL for DTS C++ Class `Hello`

```
#ifndef __hello_idl
#define __hello_idl
/*
 *
 * Generated on Tue May 16 12:08:56 1995
 * Generated from hello.hh
 * Using IBM VisualAge C Set ++ for OS/2, Version 3.00
 */
#include <somobj.idl>
interface Hello;
interface Hello : SOMObject {
    void  sayHello ();
#ifdef __SOMIDL__
    implementation {
        align=0;
        sayHello: public,nonstatic,cxxmap="sayHello()",cxxdecl="void sayHello();";
        somDefaultConstVAssign: public,override;
        somDefaultConstAssign: public,override;
        somDefaultConstVCopyInit: public,override,init;
        somDefaultInit: public,override,init;
        somDestruct: public,override;
        somDefaultCopyInit: public,override;
        somDefaultConstCopyInit: public,override;
        somDefaultVCopyInit: public,override;
        somDefaultAssign: public,override;
        somDefaultVAssign: public,override;
        declarationorder = "sayHello, somDefaultConstVAssign, somDefaultConstAssign,
somDefaultConstVCopyInit, somDefaultInit, somDestruct";
        releaseorder:
                sayHello;
        callstyle = idl;
        dtsclass;
```

```
        directinitclasses = "SOMObject";
        cxxmap = "Hello";
        cxxdecl = "class Hello : public virtual SOMObject";
        dllname = "hello.dll";
    };
#endif
};
#endif /* __hello_idl */
```

C++ Client of DTS C++ Class `Hello`

```
#include <iostream.h>
#include "hello.hh"

int main(void)
{
    Hello obj;

    obj.sayHello();
}
```

OO COBOL Client of C++ Class `Hello`

```
        Identification Division.
        Program-id. "Client".
        Environment Division.
        Configuration section.
        Repository.
            Class Hello is "Hello".
        Data Division.
        Working-storage section.
        01 obj usage object reference Hello.
        01 env usage pointer.
        Procedure Division.
            Call "somGetGlobalEnvironment" returning env.
            Display "Calling somNew".
            Invoke Hello "somNew" returning obj
            Display "Calling sayHello".
            Invoke obj "sayHello" using by value env.
            Display "Calling somFree".
            Invoke obj "somFree"
            Goback.
        End program "Client".
```

Smalltalk Client of DTS C++ Class `Hello`

```
tstHello
        "to test the SOMHello class"
        | obj |

        obj := SOMHello new.
        obj sayHello.
        obj somFree.
```

Implementation of DTS C++ Class `Hello`

```
#include <iostream.h>
#include "hello.hh"

void Hello::sayHello()
{
    cout << "Hello from C++" << endl;
}
```

The second example illustrates the use of a range of C++ class methods, such as constructors and assignment operators. The IDL generated by the DirectToSOM C++ compiler is shown following the class definition. The class `GenericString` contains a default constructor (which maps to an override or `SOMObject::somDefaultInit`), a copy constructor (maps to an override of `SOMObject::somDefaultConstCopyInit`), and a third, non-SOM, constructor that is given a SOM name of setInit. The C++ client shows these methods being used implicitly through standard C++ syntax and explicitly by calling somResolveByName, which dynamically retrieves a pointer to a given method. The class GenericString also overrides SOMObject::somDefaultConstAssign, which is called for operator= in the C++ client. Following the C++ client are examples showing how these methods are used from OO COBOL and Smalltalk. For brevity, the C++ implementation is not shown, but it would be written using standard C++ syntax.

This example also illustrates parameter passing and the use of CORBA attributes. When the SOMAttribute is applied to a C++ class data member such as length, the compiler implicitly defines and generates the methods _get_length and _set_length which retrieve and update the value of the member, respectively. This allows C++ data members to be accessed from other languages, as shown in in the OO COBOL and Smalltalk client programs. However, these data members can still be accessed using data member syntax in C++, as shown in the C++ client. Smalltalk generates the get and set methods for attributes in the Smalltalk style of member and member:, which is illustrated in the Smalltalk client program. IDL parameters can be passed as in, out, and inout. In Smalltalk, out and inout parameters are passed as arrays, where the first element in the array contains the resulting value. The methods asOUTParameter and asINOUTParameter can be used to create an array containing one object. OO COBOL has no direct support for attributes, but these methods can be called directly using their method names as shown in the OO COBOL client program. Note that OO COBOL requires that the CORBA environment parameter be passed explicitly to the target method. This parameter is passed implicitly by both DTS C++ and Smalltalk.

Definition of DTS C++ Class GenericString

```
#include <som.hh>

#pragma SOMNoMangling(on)

class GenericString : public SOMObject {
  public:
    long length;
    #pragma SOMAttribute(length, readonly)
    char *data;
    #pragma SOMAttribute(data, readonly)
    GenericString();
    GenericString(const GenericString&);
    GenericString(char *, long = -1);
    #pragma SOMMethodName(GenericString(char *, long), "setInit")
    GenericString& set(char *, long = -1);
    SOMObject *somDefaultConstAssign(somAssignCtrl *, const SOMObject*);
    void clear();
    void display();
    ~GenericString();
  #pragma SOMClassName(*, "GenericString")
  #pragma SOMIDLPass(*, "Implementation-End", "dllname = \"genstr.dll\";")
  #pragma SOMReleaseOrder( \
        length, data, \
        GenericString(char *, long), \
        set(char *, long), \
        clear(), display())
};
```

Generated IDL for DTS C++ Class GenericString

```
#ifndef __genstr_idl
#define __genstr_idl
/*
 *
 * Generated on Fri Jul  7 16:38:24 1995
 * Generated from genstr.hh
 * Using IBM VisualAge C++ for OS/2, Version 3
 */
#include <somobj.idl>
interface GenericString;
interface GenericString;
#include <somobj.idl>
```

```
interface GenericString : SOMObject {
    readonly attribute long  length;
    readonly attribute string  data;
    void  setInit (inout somInitCtrl ctrl, in string p__arg1, in long p__arg2);
    GenericString  set (in string p__arg1, in long p__arg2);
    void  clear ();
    void  display ();
#ifdef __SOMIDL__
    implementation {
        align=4;
        length:
cxxmap="length",offset=0,align=4,size=4,nonstaticaccessors,private,publicaccessors,cxxdecl="long
length;";
        data:
cxxmap="data",offset=4,align=4,size=4,nonstaticaccessors,private,publicaccessors,cxxdecl="char*
data;";
        somDefaultInit: public,override,init,cxxdecl="GenericString();";
        somDefaultConstCopyInit: public,override,init,cxxdecl="GenericString(const
GenericString&);";
        setInit:
public,nonstatic,init,cxxmap="GenericString(char*,long)",cxxdecl="GenericString(char*,long =
-1);";
        set: public,nonstatic,cxxmap="set(char*,long)",cxxdecl="GenericString& set(char*,long =
-1);";
        somDefaultConstAssign: public,override,cxxdecl="virtual SOMObject*
somDefaultConstAssign(somAssignCtrl*,const SOMObject*);";
        clear: public,nonstatic,cxxmap="clear()",cxxdecl="void clear();";
        display: public,nonstatic,cxxmap="display()",cxxdecl="void display();";
        somDestruct: public,override,cxxdecl="virtual ~GenericString();";
        somDefaultConstVAssign: public,override;
        somDefaultCopyInit: public,override;
        somDefaultAssign: public,override;
        somDefaultVAssign: public,override;
        declarationorder = "length, data, somDefaultInit, somDefaultConstCopyInit, setInit, set,
somDefaultConstAssign, clear, display, somDestruct, somDefaultConstVAssign";
        releaseorder:
                _get_length,
                s__P0,
                _get_data,
                s__P1,
                setInit,
                set,
                clear,
                display,
                length,
                data;
        callstyle = idl;
        dtsclass;
        directinitclasses = "SOMObject";
        cxxmap = "GenericString";
        cxxdecl = "class GenericString : public virtual SOMObject";
        dllname = "genstr.dll";
    };
#endif
};
#endif /* __genstr_idl */
```

C++ client of DTS C++ Class `GenericString`

```
#include "genstr.hh"
#include <iostream.h>

int main(void)
{
    cout << "create s1 statically with default constructor" << endl;
    GenericString s1;
    s1.display();

    cout << "set s1 string value" << endl;
    s1.set("string1");
    s1.display();

    cout << "create s2 statically with string constructor" << endl;
    GenericString s2("string2");
    s2.display();
```

```
    cout << "create s3 statically with copy constructor from s2" << endl;
    GenericString s3(s2);
    s3.display();

    cout << "clear s3" << endl;
    s3.clear();
    s3.display();

    cout << "assign s1 to s3" << endl;
    s3 = s1;
    s3.display();

    cout << "create obj dynamically with string constructor" << endl;
    GenericString *obj = (GenericString *)GenericString::__ClassObject->somNewNoInit();
    typedef void *(*mpt)(...);
    mpt mp = (mpt)somResolveByName(obj, "setInit");
    mp(obj, somGetGlobalEnvironment(), 0, "obj", 7);
    obj->display();

    cout << "create obj2 dynamically with copy constructor from s2" << endl;
    mp = (mpt)somResolveByName(obj, "somDefaultConstCopyInit");
    GenericString *obj2 = (GenericString *)GenericString::__ClassObject->somNewNoInit();
    mp(obj2, 0, s2);
    obj2->display();

    delete obj;
    delete obj2;
}
```

Smalltalk client of DTS C++ Class `GenericString`

```
tstString
        "to test the SOMGenericString class"
        | obj1 obj2 obj3 ctrl |

        obj1 := SOMGenericString new.
        obj1 set: 'Smalltalk object 1' pArg2: -1.
        Transcript cr; show: 'After init & set: obj1 data is ''' , obj1 data, ''''.

        ctrl := Array new: 1.
        SOMGenericString somGetInstanceInitMask: ctrl.
        obj2 := SOMGenericString somNewNoInit.
        obj2 somDefaultCopyInit: ctrl fromObj: obj1.
        Transcript cr; show: 'After somDefaultCopyInit obj2 data is ''' , obj2 data, ''''.

        SOMGenericString somGetInstanceAssignmentMask: ctrl.
        obj2 set: 'Assigned from Smalltalk object 2' pArg2: -1.
        obj1 somDefaultConstAssign: ctrl fromObj: obj2.
        Transcript cr; show: 'After somDefaultConstAssign obj1 data is ''' , obj1 data, ''''.

        SOMGenericString somGetInstanceInitMask: ctrl.
        obj3 := SOMGenericString somNewNoInit.
        obj3 setInit: ctrl pArg1: 'Smalltalk object 3'  pArg2: -1.

        obj1 somFree.
        obj2 somFree.
        obj3 somFree.
```

OO COBOL client of DTS C++ Class `GenericString`

```
        Identification Division.
        Program-id. "Client".
        Environment Division.
        Configuration section.
        Repository.
            Class GenericString is "GenericString".
        Data Division.
        Working-storage section.
        01 env usage pointer.
        01 str1 usage object reference GenericString.
        01 str2 usage object reference GenericString.
        01 str3 usage object reference GenericString.
        01 objp usage object reference GenericString.
        01 txt pic X(25) value "string1 from COBOL".
```

```
01 len pic s9(9) comp.
01 p usage pointer.
01 nullp usage pointer value null.
01 somNewNoInit pic x(12) value "somNewNoInit".
01 mp usage procedure-pointer.
01 b pic s9(9) binary.
Linkage section.
01 txt2 pic X(25).
01 envMaj pic s9(9) usage binary.
Procedure Division.
    Call "somGetGlobalEnvironment" returning env.
    Set Address of envMaj to env.
    Display "Calling somNew".
    Invoke GenericString "somNew" returning str1
    If envMaj not = 0
        Perform error-handler.

    move length of txt to len.
    set p to address of txt.
    Display "Calling Set".
    Invoke str1 "set" using by value env by value p by value len returning objp.
    Display "Calling _get_data".
    Invoke str1 "_get_data" using by value env returning p.
    Set Address of txt2 to p.
    Display txt2.

    Display "Calling somNewNoInit".
    Invoke GenericString "somNewNoInit" returning str2.
    Display "Calling somDefaultConstCopyInit".
    Invoke str2 "somDefaultConstCopyInit" using by value nullp str1.
    Invoke str2 "display" using by value env.

    Move "assign string" to txt.
    set p to address of txt.
    Invoke str1 "Set" using by value env by value p by value len returning objp.
    Display "Calling somDefaultAssign".
    Invoke str2 "somDefaultAssign" using by value nullp str1 returning objp.
    Invoke str1 "display" using by value env.

    Display "Calling somNewNoInit".
    Invoke GenericString "somNewNoInit" returning str3.

    Move "setInit string" to txt.
    set p to address of txt.
    move 14 to b.
    Display "Calling setInit".
    Invoke str3 "setInit" using by value env nullp p b.
    Display "Calling display".
    Invoke str3 "display" using by value env.

    Display "Calling somFree".
    Invoke str1 "somFree".
    Invoke str2 "somFree".
    Invoke str3 "somFree".
    Goback.

error-handler.
    call "somExceptionId" using by value env returning p.
    Set Address of txt2 to p.
    Display "major: " envMaj.
    Display "error: " txt2.
    call "somExceptionFree" using by value env.
    Goback.
End program "Client".
```

9. CORBA and DSOM

The IDL generated by the DirectToSOM C++ compiler is CORBA compliant, and can be used in a non-SOM environment. While additional SOM-specific information is generated, it is guarded by the `__SOMIDL__` macro, allowing the files to be used in a non-SOM environment.

While the examples have not been tested in a distributed SOM (DSOM) environment, they should work, as successful testing has been performed with DirectToSOM C++ and DSOM using the IDL generated by the DirectToSOM C++ compiler. Note that additional information can be supplied in the generated IDL for DSOM support using the `SOMIDLDecl` and `SOMIDLPass` pragmas. `SOMIDLDecl` allows the programmer to specify the IDL declaration that should be generated for a given artifact (for example, a type or a member function). This is useful, for example, in specifying the direction of parameters, which default to `inout` for address parameters and `in` otherwise. The `SOMIDLPass` pragma allows arbitrary strings to be generated into the IDL. Further details on the IDL generation performed by the compiler, including the mapping of C++ types to IDL types, and DSOM support with DirectToSOM C++ can be found in [5].

10. Restrictions and Semantic Differences

In general, DirectToSOM C++ supports most of the standard C++ constructs, including templates and pointers to members. This section covers the major restrictions when using DirectToSOM C++ and the semantic differences between standard and DirectToSOM C++ classes.

- class hierarchy:
  A class hierarchy must contain all DirectToSOM or all standard C++ classes; a mixed hierarchy is not supported. In addition, only a single occurrence of each non-virtual base class is allowed within a DirectToSOM hierarchy. `SOMObject` is a special case, as it is implicitly treated as virtual.

- inline member functions
  Inline member functions are currently generated out-of-line by the compiler so that RRBC won't be compromised. For example, if an inline member function accessed a private data member, any client in which that member function were inlined might require recompilation if a new private data member were added.

- `sizeof`
  Because the size of a DirectToSOM C++ class is not known until run time, `sizeof` is a run-time constant expression for DirectToSOM instances and is not allowed in contexts that require compile-time evaluation. For example, given a SOM class `A`, the declaration of an integer initialized to the size of the type `A` is correct because the size of type `A` can validly be determined at run time in this context. But according to the C++ language rules, an array bound must be a positive integral constant expression, so `sizeof(A)` is not valid in supplying the bound of an array because the compiler cannot evaluate the expression until run time. Note that the value returned by `sizeof` is fixed within a given execution.

- `offsetof`
  `offsetof` depends on the data member access, due to the reordering of data members to enable RRBC (see [4] or [5] for details). Public data starts at offset 0, while protected and private together start at offset 0. In addition, the offset is always relative to class that introduces it. `offsetof(base, base_element)` is always equal to `offsetof(derived, base_element)`. Therefore, you cannot use the offset of a data member within a class to get to the beginning of the instance data.

- data member addresses
  The addresses of nonstatic RRBC data members are not contiguous within a class, one of the few areas where DirectToSOM C++ does not conform to the C++ standard. This is necessary because the member is implemented and treated as a reference, so when you take the address of an RRBC instance, you get the value of the reference, which is the address of the underlying hidden structure.

- initializer list
  You cannot use an initializer list to initialize DirectToSOM objects, because DirectToSOM classes always inherit from the `SOMObject` class. (An initializer list cannot be used to initialize an object of a class that has a base class).

- calling through a NULL pointer
  You cannot call a nonvirtual function through a NULL pointer to an RRBC instance, because the method routing must be performed through the SOM API using a valid class instance.

- casting
  You cannot cast a pointer-to-RRBC object to arbitrary storage or an unrelated type (class punning), because the SOM API depends upon the underlying object layout.

- linking
  All static data and member functions must be defined by link time because they are used to construct the class tables.

11. Usage Considerations

Apart from the issues raised above, one of the major considerations when deciding to make a class a DirectToSOM class is performance. While DirectToSOM C++ gives you the power and flexibility of the SOM language-neutral object model, there is an overhead in using it. For interfaces involving many calls to very simple methods, this overhead can be noticeable. However, much ofr the overhead is not intrinsic to the SOM model, and will probably be alleviated in future releases of the product.

In terms of run-time performance, every SOM virtual method invocation currently requires an indirection through a pointer in the SOM class data structures to invoke the method. This is in the form of a call through a function pointer to a *thunk*, which is a small code sequence that performs a few setup instructions and branches to the target method. Nonvirtual and static member functions are also currently called through a function pointer.

In addition, instance data access, either from the client or within the implementation, currently requires a *data token* function pointer call through the SOM class data structures to retrieve addressability to that data. For access through the `this` pointer, one call is required within each method invocation. (For CORBA attributes, this calls is in addition to the `_get`/`_set` method call, if any is used, because the target method must also access the instance data through the data token. Note that, as mentioned earlier, instance data can be made into attributes to provide external access, but can still be accessed from DirectToSOM C++ directly through the data token without calling the `_get`/`_set` methods). A future release of DirectToSOM C++ will probably lessen this overhead by accessing instance data for the `this` pointer through a supplied register value. Backend support for specifying that the result of the data token function call as invariant can also improve performance by allowing optimizations such as hoisting data token calls out of loops.

With respect to run-time storage requirements, each SOM object contains a pointer at the beginning to the class run-time data structures. So, when a class is made into a SOM class, each object will increase in size by the size of the pointer.

In certain situations, defining a class as a SOM class can also result in an increase in the static object size. Because inline methods are currently generated out-of-line, the object size may increase noticably over using native C++ for classes with a large number of inline methods. In addition, prolog code is currently generated for each constructor to handle the initialization control vector that prevents multiple base class initialization. For classes with a large number of constructors, this additonal code can be discernable in the resulting object size.

In most cases, however, there will be a certain set of classes that are exposed in the interface, for which language neutrality support is required. Usually, there will be many more internal classes that do not require this additional functionality. Thus DirectToSOM and non-DirectToSOM classes can be mixed as the needs of the application dictate.

As with using C++ instead of C, or a high-level language instead of assembler, using DirectToSOM C++ for language neutrality support involves a tradeoff between programmer productivity and program efficiency. You can choose to manage language neutrality explicitly and probably produce faster code, but this gains will probably be acheived at the cost of language restrictions and increased program management, which can be both tedious and error-prone. Note also that there are additional advantages to using DirectToSOM C++ over just the language neutrality support, such as release-to-release binary compatibility (RRBC) and distributed object support (DSOM). For further information, see [4] and [5].

12. Conclusion

The SOM support for all three languages, C++, Smalltalk, and OO COBOL, is relatively natural and intuitive. In particular, the wrapper classes make the Smalltalk mapping to SOM almost invisible. The SOM model clearly provides a language-independent object model that solves the interlanguage object sharing problem in an almost seamless fashion. Although the examples in this paper are relatively simple in nature, they illustrate that a variety of C++ language features can be used effectively in a mixed language environment. It is therefore quite feasible that a C++ class library could be ported to DTS C++ and used from other languages through SOM.

References

[1] ANSI COBOL-97 Working Paper, December 20, 1994.

[2] *The Common Object Request Broker: Architecture and Specification*. Farmingham, MA: Object Management Group, 1992.

[3] Danforth, S., P. Koenen, and B. Tate, *Objects for OS/2*. New York: Van Nostrand Reinhold, 1994.

[4] Hamilton, J. "Reusing Binary Objects with DirectToSOM C++." *C++ Report*, March 1996.

[5] Hamilton, J. *Programming with DirectToSOM C++*. New York: John Wiley and Sons, to be published in 1996.

[6] Hamilton, J., R. Klarer, M. Mendell, B. Thomson, "Using SOM with C++", *C++ Report*, July/August 1995.

[7] *IBM VisualAge C++ for OS/2 Version 3.0 Programming Guide.* IBM Document S25H-6958, 1995.

[8] *IBM VisualAge for COBOL for OS/2 Version 1.1 Programming Guide*. IBM Document SC26-8423, 1995.

[9] *IBM VisualAge SOMSupport Guide*, 1994. (Online document supplied with IBM VisualAge for Smalltalk for OS/2 Version 2.0).

[10] *SOMObjects Base Toolkit User's Guide Version 2.0*. IBM Document SC23-2680, 1993.