Release-to-Release Binary Compatibility and the Correctness of Separate Compilation

Ira R. Forman
Michael H. Conner
Scott H. Danforth
Larry K. Raper

IBM Object Technology Products
11400 Burnet Road
Austin, Texas 78758

Introduction

Despite all of its promise, software reuse in object-oriented programming has yet to reach its full potential. We recognize that a major impediment to reuse is the inability to evolve a compiled class library without abandoning the support for the already compiled applications. The underlying cause of this problem is that the typical object-oriented model has elements that are not part of the interface model of the linkage editor/loader. Therefore, an object-oriented model must be carefully designed so that class-library transformations that should not break already compiled applications, indeed, do not break such applications. This leads us (at the conclusion of this paper) to a new criterion for the correctness of separate compilation for all programming systems.

The System Object Model (SOM) is so designed. The next section gives an overview of SOM, after which we return to the problem of producing and supporting release-to-release binary compatible class libraries. This paper presents the SOM solution to this problem.

The SOM Model

In SOM [2], classes are objects whose classes are called metaclasses. A class is different from an ordinary object because a class has (in its instance data) an instance method table defining the methods to which

instances of the class respond. During the initialization of a class object, a method is invoked on it that informs the class of its parents. This allows the class to build an initial instance method table. Once this is done, other methods are invoked on the class to override inherited methods or add new instance methods.

When diagraming class hierarchies, this paper uses the convention that metaclasses are drawn with three concentric circles, ordinary classes (i.e., classes that are not metaclasses) are drawn with two concentric circles, and ordinary objects (i.e., objects that are not classes) are drawn with a single circle. The initial state of an example SOM program is depicted in Figure 1. There are four objects SOMObject (a class), SOMClass (a metaclass), Dog (an ordinary class), and Rover (an ordinary object). There are two relations among objects that one must understand.

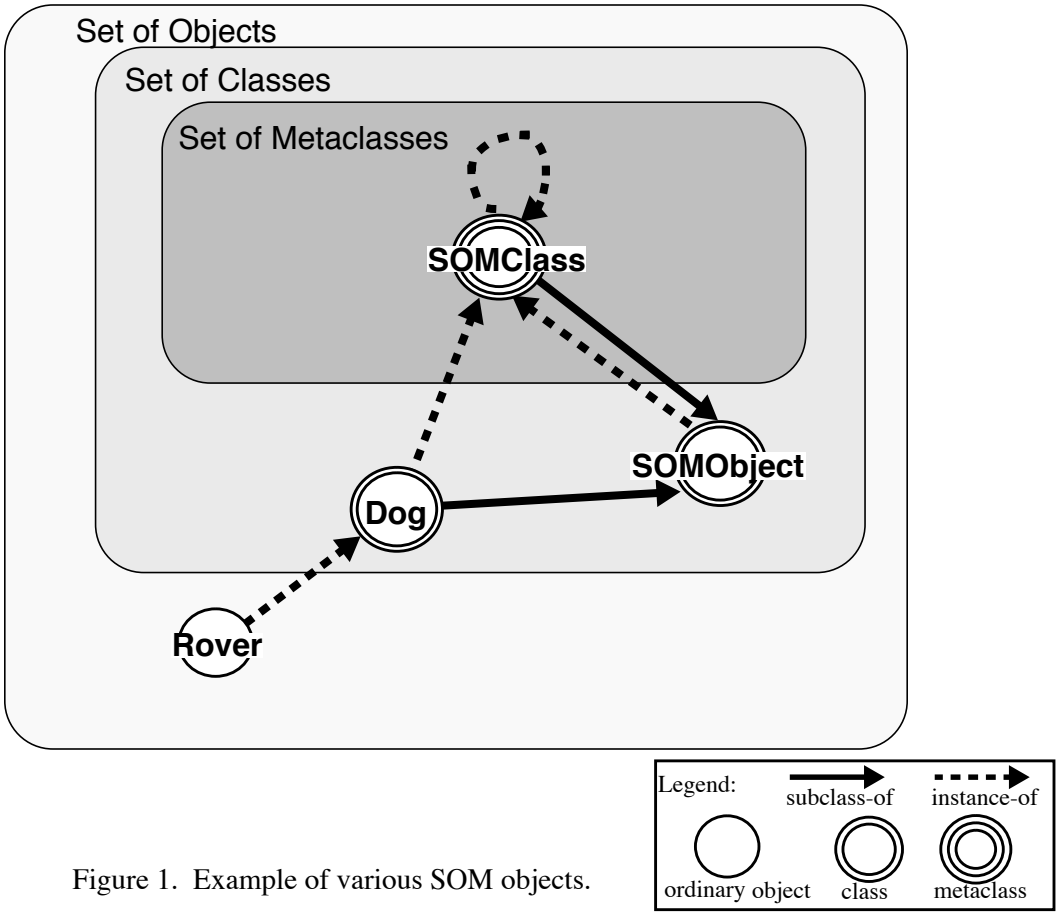

Figure 1. Example of various SOM objects.

First, there is the *instance of* relation between objects and classes depicted by the dashed arrow from an object to its class. When convenient the inverse relation, class of, is also used. SOMObject is an instance of

SOMClass and SOMClass is the class of itself. An object's class is important because an object responds only to the methods that are supported by its class (that is, the methods that the class introduces or inherits).

Second, there is a relation between classes called the *subclass of* relation, which is depicted by the solid arrow from a class to each of its parents. SOMClass is a subclass of SOMObject. SOMObject has no parents.

SOMObject introduces the methods to which all SOM objects respond. As a subclass of SOMObject, SOMClass is an object but in addition introduces the methods to which all classes respond. For example, SOMClass introduces the somNew method, which creates instances of a class. Also, the methods responsible for creating and modifying instance method tables are introduced. All metaclasses in SOM are ultimately derived from SOMClass. (Similar arrangements of classes are also used in other object models, for example, CLOS[9] and Dylan[1].) SOMClass and SOMObject are the two most important classes of the SOM kernel.

Interfaces to SOM objects are described using IDL, an object interface definition language defined by the Common Object Request Broker Architecture (CORBA [10]) standard of the Object Management Group (OMG). SOM IDL is a CORBA-compliant version of IDL used to allow SOM class descriptions to be supplied in addition to object interface definitions. (That is, the interface to a class is described by the IDL alone, SOM IDL allows additional information about the implementation to be added.) The SOMobjects Toolkit has tools called emitters that translate SOM IDL into language-specific bindings for the corresponding classes of SOM objects (e.g., for C programmers this means that emitters produce header files for both the users of the class and the implementor of the class). In addition, there are C++ compilers that produce code that directly uses the SOM runtime [8].

Below is the basic structure of an IDL definition for an object interface named Dog. At the same time, it is a SOM IDL description of a class Dog that supports this interface. The `#ifdef` and `#endif` (which, for simplicity, are omitted from subsequent examples) are part of the IDL language and are used to hide the SOM class implementation section from non-SOM IDL compilers.

```
interface Dog : SOMObject {
  <method and attribute declarations here>
  #ifdef __SOMIDL__
  implementation {
    metaclass = SOMClass;
    <instance variable declarations here>
  };
  #endif
};
```

In this example the interface Dog inherits from the SOMObject interface, and at the same time, the class Dog is declared to be a subclass of SOMObject. CORBA and SOM support multiple inheritance; additional parents of Dog can be listed alongside SOMObject in a comma-separated list. The actual methods and instance variables of Dog are not relevant to the current discussion. As illustrated here, the implementation section can explicitly indicate a metaclass to be associated with the class of objects that support the interface being defined. This association is not necessarily direct, however. For reasons that will become clear, the actual class of the class described by any given SOM IDL is, in general, a subclass of the indicated metaclass.

The Library Compatibility Problem

With SOM there is an important distinction between the term API (application programming interface) and ABI (applications binary interface). The API is the interface that a programmer uses, while the ABI refers to the specific conventions on which running programs depend. API compatibility ensures that applications can be recompiled successfully; ABI compatibility ensures that compiled applications continue to run successfully.

Let us consider the predicament of a software vendor that is selling a software library; the customers of the library vendor use the library to develop applications. The library vendor gives the customers a description of the API to the library (e.g., header files or OMG IDL) and the compiled library in the form of a dynamically linked library (DLL). The interface to the (compiled) DLL is called the ABI. The advantage of using a DLL (to both the library vendor and the application producer) is that multiple applications may run with one copy of the DLL present, which vastly reduces the memory requirements and improves response time.

A problem arises when the library vendor wishes to evolve the library. The library vendor must maintain release-to-release binary compatibility, because it is impossible for the application producer to recompile the already distributed copies of the application.

The problem is couched in the terms of a library vendor, an application vendor, and dynamically linked libraries, because we wish to emphasize the relevance of our work to this area. In reality, the problem and our solution are much more general. The granularity goes down to the individual programmer who provides libraries for others or even herself. Whenever a library is changed one must consider the impact on the applications that are already using the library.

Pragmatically, a revision to a library is release-to-release binary compatible if all the applications that depend on the library still work. Because of a lack of precise specifications for both the library and the application, this is a subjective judgement. For example, one can repair a defect in the library only to find that application programmers consider it a feature.

Now this a bit abstract; let's take as an example how C++ fails to support release-to-release binary compatibility. Figure 2 depicts the situation where class X (from the class library) is subclassed in the application with class Y. With release 1.0 of the library, the application runs perfectly; in particular, fmethod can be invoked on iY, an instance of class Y. Now let us consider what happens when the library vendor makes the seemingly innocuous change of adding a new method (hmethod) to class X, which is invoked from fmethod. Now because the application was built with the old library definition, the method table for Y has a pointer to gmethod in its second entry. However, when the fmethod is invoked on iY, it tries to subsequently invoke hmethod, which in class X is the second entry in the method table. The result is that gmethod is called when hmethod was specified.

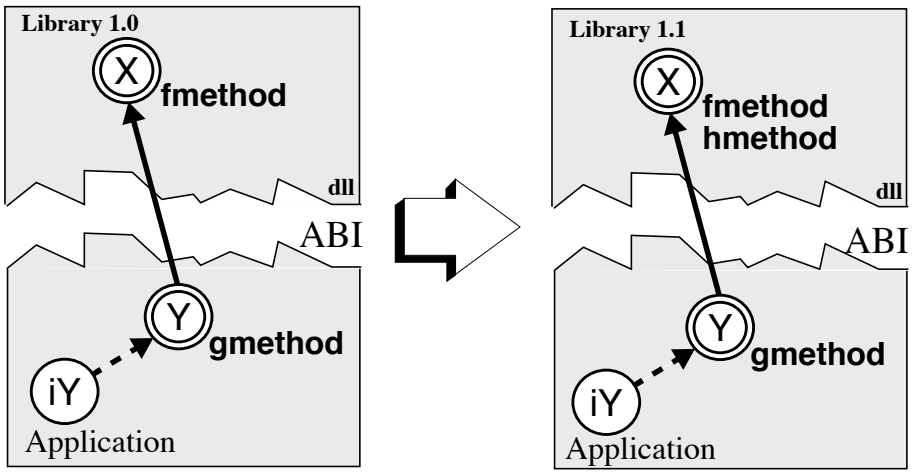

Figure 2. An incompatibility produced by the typical C++ compiler.

This situation is both familiar and frustrating to the users of C++. What makes this problem even more insidious is the fact that it appears that the problem is in class X, which is called the base class in C++. Because of this example and others like it, the myth of the "fragile base class problem" has arisen. The poor base class is blamed, diverting our attention from the real problem: the compiler/linker combination does not properly support subclassing across the binary interface.

Defining Release-to-Release Binary Compatibility

Creating an effective definition of release-to-release binary is not simple. Consider this direct attack on the problem. Let

A ⋈ L mean application A is combined to library L

and let

SAT(P, S) mean program P satisfies specification S

where P, A, and L represent compiled modules.

Straw-man Definition 1. L_1 is a *RRBC-successor* to L_0 if

$$SAT(A \bowtie L_0, S) \text{ implies } SAT(A \bowtie L_1, S)$$

for all reasonable functional specifications S and for all applications A using library L.

The problems with this definition characterize the difficulties of supporting the evolution of binary class libraries. First, the operator ⋈ symbolizes many different implementations (one for each compiler/

linker/loader combination). Second, there is no satisfactory definition of "reasonable functional specification." Third, establishing the implication of the definition is tantamount to proving program equivalence, an unsolvable problem. Fourth, even if program equivalence were not unsolvable, defect removal implies some functions of the new library are not equivalent. Fifth, even if "reasonable functional specification" could be satisfactorily defined, real software products rarely have satisfactory formal specifications. Sixth, for a generally available class library product, one cannot know the set of applications using the library (this simply means that for generally available libraries, a successor must support all possible applications).

In light of the above difficulties, the only recourse is to develop an engineering discipline for software libraries. This discipline should be based on transformations that are guaranteed to be compatibility preserving. This means that a library revision is compatible if only these transformations are used to derived it from the old library. The engineering discipline, then, requires a careful justification for those revisions that are not attained by the transformations.

The goal of this engineering discipline for compiled libraries can be stated as:

Only application alteration necessitates recompilation

This implies that if the evolution of the class library does not require changes to the application source, then the application should not require recompilation.

In terms of our straw-man definition, these transformation produce compatible RRBC-successors. But because the straw-man definition is not formal, each transformation must be independently justified. In addition, enumerating the transformations of this discipline is not enough. Because compiled libraries must be accommodated, we must require that the technology for binding applications to libraries must support the transformations. Now as we shall see, for procedural programming, current linkage editors are adequate, but for object-oriented programming, SOM provides the most complete set of transformations.

Procedural Programming

For procedural programming (the style that preceded object-oriented programming), the constituents of an applications programming interface are procedures as depicted in Figure 3. Applications make procedure calls and linkage editors ensure that each procedure call is bound to the appropriate procedure implementation.

With this ABI, the problem of release-to-release binary compatibility reduces to the question of:

Is each procedure of the new library a more complete implementation of its predecessor?

This implies that there are five transformations available to our engineering discipline:

Transformation 0: The procedure can be reimplemented to provide better performance for the same functional interface. (Note that we ignore any pathological real-time situation where a better performing implementation fails to meet its specification.)

Transformation 1: The domain of the procedure can be enlarged to return values for inputs for which it previously aborted, failed to return (infinite loop or deadlock).

Transformation 2: On systems where the calling conventions indicate the number of parameters, the number of parameters of a procedure can be increased.

Transformation 3: Addition of new procedures.

Transformation 4: Retraction of private procedures.

Our engineering discipline says that as long as only these transformations are applied the new procedure library is an release-to-release binary compatible revision of the old library. Of course, there are good reasons for making changes that are outside these transformations; these must be carefully analyzed for their impact on applications using the old libraries.

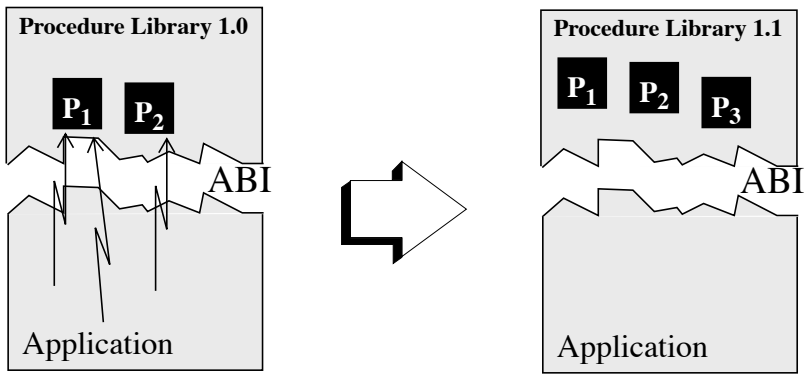

Figure 3. Procedure libraries evolve safely by adding new procedures.

Object-Oriented Programming

The richness of Object-Oriented Programming adds new facets to the ABI. Besides the ABI of procedural libraries, OOP applications subclass the classes of the library (see Figure 4). The new aspects of the problem give rise to a new engineering discipline needed to ensure release-to-release binary compatibility of a class library and thus, the continued functioning of the dependent applications.

The additional transformations required to support our engineering discipline are:

Transformation 5: Addition of new instance variables to objects
Transformation 6: Addition of new methods to classes
Transformation 7: Insertion of new classes into the hierarchy
Transformation 8: Migration of a parent class downward in the class hierarchy
Transformation 9: Migration of a method upward in the class hierarchy
Transformation 10: Retraction of private classes.
Transformation 11: Retraction of private methods
Transformation 12: Retraction of private instance data
Transformation 13: Reorder the methods of a class
Transformation 14: Reorder the instance variables of an object

The need for these additional transformations is directly caused by permitting subclassing across the ABI.

SOM is designed to support these additional transformations.

When classes are first class objects

SOM allows and encourages the definition and explicit use of metaclasses. In this wider ABI, our engi-

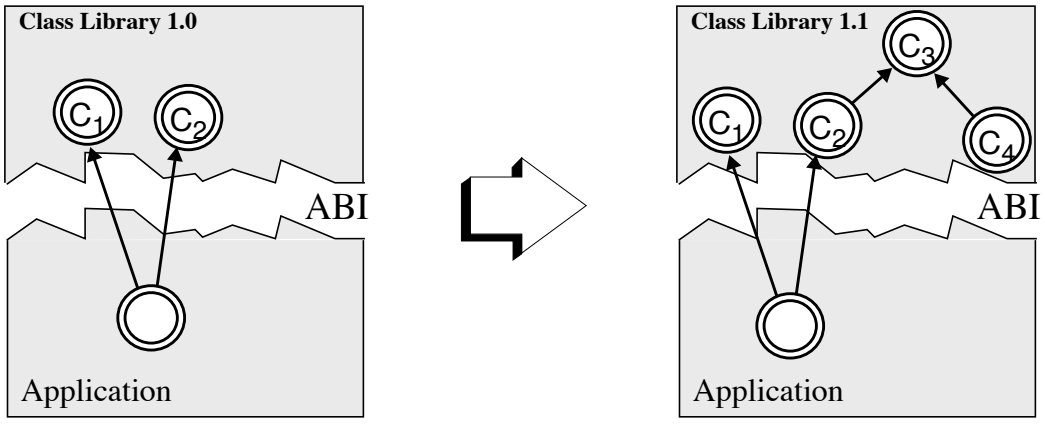

Figure 4. Applications subclass from class libraries which evolve by adding new classes.

neering discipline requires an additional transformation.

Transformation 15: The class of a class (i.e., its metaclass constraint) can be moved downward in the class hierarchy.

This transformation is similar to Transformation 8 in that when migrating the metaclass downward the new metaclass supports all the functionality of the old metaclass. However, there is a constraint on the appropriateness of a new metaclass.

Consider the simple single-inheritance example illustrated by Figure 5. In this figure, X is an instance of XMeta; we assume that XMeta supports a method bar and that X supports a method foo that uses the expression bar(class(self)). That is, the method foo invokes a method on the class of the object on which foo is operating. Now consider what happens when X is subclassed by Y, a class that has an explicit metaclass declared in its SOM IDL as in Figure 5. If the class hierarchy were to be formed as in Figure 5, then an invocation of foo on an instance of Y would fail because YMeta does not support bar. This situation is referred to the as metaclass incompatibility.

SOM does not allow hierarchies with metaclass incompatibilities. Instead, SOM builds derived metaclasses that prevent this problem from occurring. The actual SOM class hierarchy that results for Y is

```
interface X {
...
  void foo();
  implementation{
    metaclass = XMeta
  };
};
where
   foo()
   {...
    bar(class(self));
    ...};

interface Y:X {
...
  implementation{
    metaclass = YMeta
  ...
```

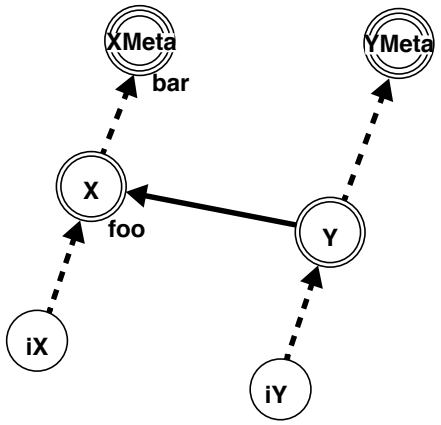

Figure 5. Example of a metaclass incompatibility.

depicted in Figure 6, where SOM has automatically built the metaclass DerivedMetaclass; this ensures that the invocation of foo on instances of Y do not fail. This example shows that the metaclass statement in the SOM IDL is treated as a constraint on the actual metaclass. The derived metaclass can be viewed as the minimal metaclass supporting the constraints of metaclass compatibility.

This situation is called *the metaclass compatibility problem* [7]. In SOM there is no such problem; in a situation where the explicitly declared metaclass is not compatible with the parents of the class, an appropriate metaclass is constructed -- this is the derived metaclass. Because class construction is a dynamic activity in SOM, this derivation is actually accomplished at runtime with no need for prior description in IDL.

Now all languages have to solve the metaclass incompatibility problem. For example, C++ solves the problem by not having metaclasses. Smalltalk solves the problem by not allowing metaclasses to be explicitly named. CLOS has a rule for checking compatibility at class construction time that eliminations the possibility of a metaclass incompatibly (but failure of the check causes a runtime failure). Only SOM handle metaclass incompatibility without restricting the programmer.

How Derived Metaclasses Support Binary Libraries

SOM relieves programmers of the responsibility for getting the metaclass right when defining a new class. At first glance, this might seem to be merely a useful (though very important) convenience. But, in fact, it is essential to the support of release-to-release binary compatible libraries in SOM. Although a programmer

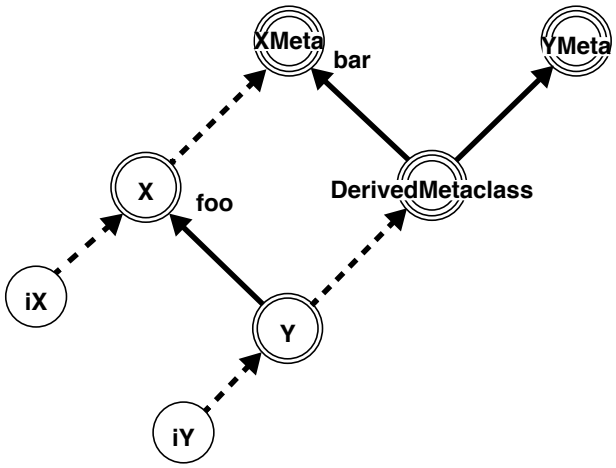

Figure 6. Example of a derived metaclass.

might, at one time, know the metaclasses of all classes above a new subclass, and, as a result, be able to explicitly derive an appropriate metaclass for the new class, SOM must guarantee that this new class still executes correctly when any of its ancestor class's implementations are changed (and this could include a choice of different metaclasses). Thus, a SOM programmer never needs to consider a newly defined class's ancestors' metaclasses. Instead, explicit metaclasses should only be used to add in desired behavior for a new class. Anything else that is needed is done automatically.

Figure 7 contains a specific example. The application and the two libraries in the upper part of the diagram work correctly together. If the Library A evolves by inserting a new metaclass (which should be acceptable as it is moving the metaclass constraint downward), the metaclass incompatibility depicted in Figure 5 is attained.

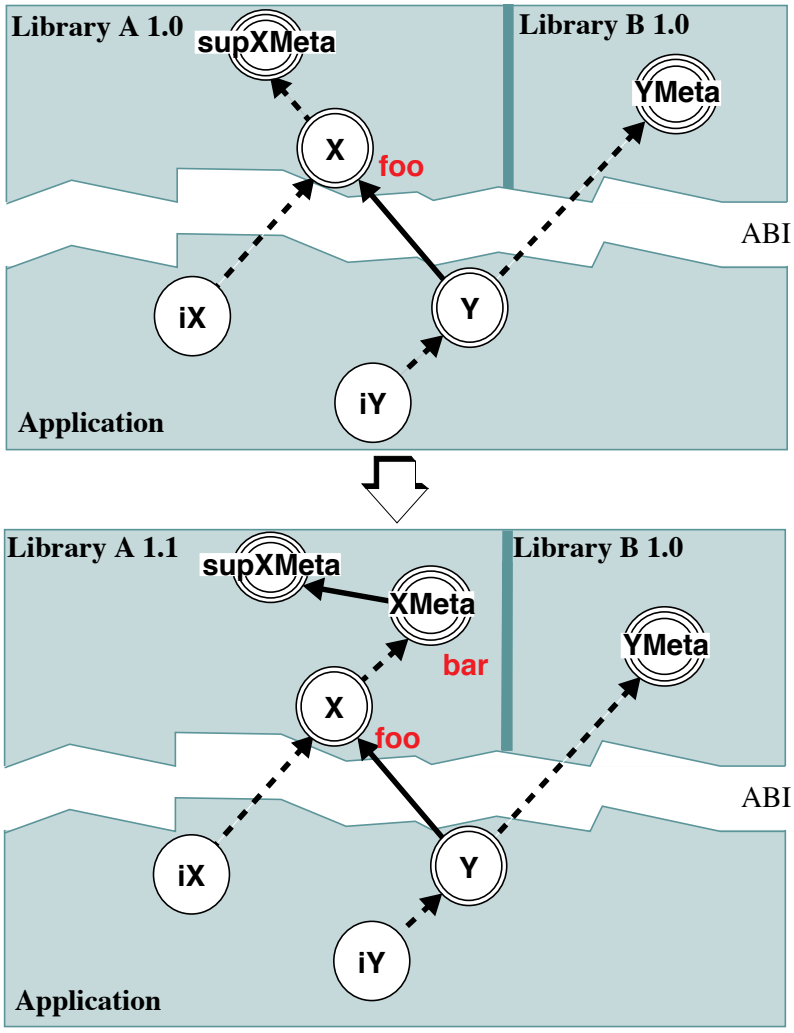

Figure 7. Example of a metaclass incompatibility arising in library use.

Figure 8 shows how the lower half of Figure 7 looks when built dynamically by the SOM kernel. This example makes one further point. The metaclass incompatibility can arise across libraries. There is no way for a metaclass programmer to know about how metaclasses are used in applications. Without the notion of the derived metaclass, there is no way for the application programmer to avoid the situation. Yet it is very valuable to compose metaclasses as is shown in [4].

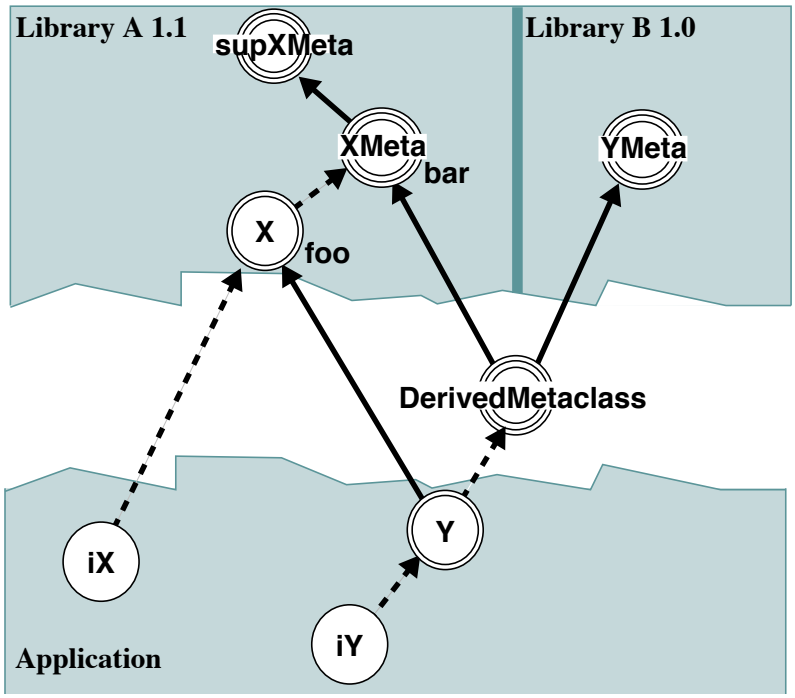

Figure 8. SOM prevents the Metaclass Incompatibility.

A comparison of support in several object models

Now when choosing a programming system in which to produce compiled libraries one needs to consider which transformations are supported. Table 1 gives such a comparison for several object models[1]. The Smalltalk and C++ columns are generic, but the Delta/C++ refers to the C++ compiler from Silicon Graphics [12] and OBI refers to the research work of Sun Microsystems [5].

In Table 1, ✔ means the transformation is supported and ✖ means it is not. In the cases where the transformation has no meaning to that technology, Table 1 has an “n/a” entry.

1. We exclude Microsoft’s COM [11] because it is an interface model, not an object model and it’s ABI forbids subclassing between library and application. If our analysis technique is applied to COM, one sees that it supports only Transformations 0, 1, 3, and 4, which places it in the category of procedural programming rather than object-oriented programming.

Table 1: Comparison of Support for Compiled Class Libraries

Transformation	Compiled Smalltalk	Compiled CLOS	Generic C++	SOM	Delta/ C++	OBI	Objective -C	Java
0: improve performance	✔	✔	✔	✔	✔	✔	✔	✔
1: eliminate failures	✔	✔	✔	✔	✔	✔	✔	✔
2: add parameter	✖	✖[a]	✖[b]	✖[c]	✖	✖	✖	✖[b]
3: add procedure	n/a	✔	✔	✔	✔	✔	✔	n/a
4: retract private procedure	n/a	n/a	✔	✔	✔	✔	✔	n/a
5: add instance variable	✖[d]	✔	✖	✔	✔	✔	✖	✔
6: add new method	✔	✔	✖	✔	✔	✔	✔	✔
7: insert new class	✔	✔	✖	✔	✔	✔	✖	✔[e]
8: migrate parent downward	✔	✔	✖	✔	✔	✖[f]	✖	✔[d]
9: migrate method upward	✔	✔	✖	✔	✔	✔	✔	✔
10: retract private class	n/a	n/a	✖	✔	✔	✔	✔	✔
11: retract private method	n/a	n/a	✖	✔	✔[g]	✔	n/a	✔
12: retract private data	n/a	n/a	✖	✔	✔	✔	n/a	✔
13: reorder methods	n/a	✔	✖	✔	✔	✖[f]	✔	✔
14: reorder instance variables	✖	✔	✖	✔	✔	✖[f]	✖[h]	✔
15: migrate metaclass constraint downward	✖	✖	n/a	✔	n/a	n/a	✖	n/a

a. This transformation is supported for functions in compiled LISP, but is not for generic methods in CLOS.
b. Because of overloading in C++, this box gets a formal ✖, because adding new parameters is defining a new method. One might think this is not a problem, because overloading allows multiple procedures with the same name. However, this causes an answer for procedures that is different than that for methods.
c. However, SOM does support procedures and methods that are defined to have a variable number of parameters.
d. With compiled Smalltalk, addition of instance variables requires recompilation of applications.
e. Current implementations of Java do not support these, but the new specification make it clear that all of our transformations must be supported (see [6] pages 206-214).
f. One should bear in mind that OBI is a research project that had the additional goal of supporting multiple versions of a class.
g. Because this is a C++ approach, one must ensure that private members that have not been exposed by friend specifications.
h. Objective-C supports direct access interface to instance data making this box ✖. Brad Cox informs us that wise Objective-C programmers avoid use of this facility (and by doing so, turn this box to a ✔.)

Note that none of these technologies (including SOM) support Transformation 2. This is not an impediment to evolving a class library, because one can define a new method (with a new name) having the expanded signature while retaining the old method. Now, the compiled applications run as expected while new applications use the new method.

Our presentation has been informal; e.g., we have not defined the criteria for a complete set of transformations that are compatibility preserving (for that matter, neither has compatibility preserving been formally defined). Usually lack of completeness of a transformation set implies lack of sufficiency. For the problem of evolution of class libraries, this is not the case. Clearly, SOM is not complete, because Transformation 2 is not supported. But as we argue above, this is not a serious impediment to evolving a class library.

Experience

The SOM technology described here is more than a theory; it has been heavily used both within IBM and throughout the OS/2 development community for over three years. Numerous OS/2 applications and system facilities such as IBM's LAN Server, MMPM/2, Ultimail, IBM Works, and even the OS/2 Workplace Shell (the desktop user interface for OS/2) are built on the SOM technology. Even moderately-sized applications such as these contain dozens, and in some cases hundreds of SOM classes. These applications were developed independently and have evolved, even while SOM itself has been evolving.

Figure 9 shows the progression of the SOM technology and the OS/2 operating system. The debut of SOM 1.0 occurred in April 1992; it was an essential part of the first 32-bit OS/2 2.0 release. Although the 1.0 level of SOM provided only a basic single-inheritance programming capability most of the RRBC features described in this paper were already present. In fact, it was the very presence of the RRBC features in the

early SOM that permitted the technology to evolve.

The 2.0 level of SOM made a significant leap in terms of new features and new capabilities. This was the first attempt of any commercial product to embrace the fledgling CORBA 1.1 standard. For SOM this meant a new definition language for SOM interfaces, extensions to the base object model itself (such as multiple inheritance), and changes in the infrastructure to support distribution via an Object Request Broker (ORB), known as DSOM. Because the product cycles of the OS/2 and SOM development teams were entirely independent, SOM 2.0 appeared in a separately licensed product called the SOMobjects Developer Toolkit. In addition to offering the basic SOM programming tools the toolkit also included a broad set of application frameworks and sample programs. When installed on any OS/2 2.0 or 2.1 system, the SOM toolkit completely superceded the earlier level of SOM still found there. So, although none of the OS/2 SOM applications yet took advantage of the new features of SOM 2.0, all of the frameworks derived from SOM continued to work without recompilation, while newly developed frameworks exploiting SOM 2.0 ran in the same environment.

In October of 1994, SOMobjects Toolkit release 2.1 became generally available; this release of SOM corrected some reported defects (the SOM developers are no more perfect or omniscient than anyone else) and added many performance enhancements. This time, the OS/2 Warp product cycle and the SOM product cycle coincided more favorably and the 2.1 level of SOM was included as an integral element of Warp. The most significant thing to note is that the OS/2 Warp Workplace Shell started its development (in Boca

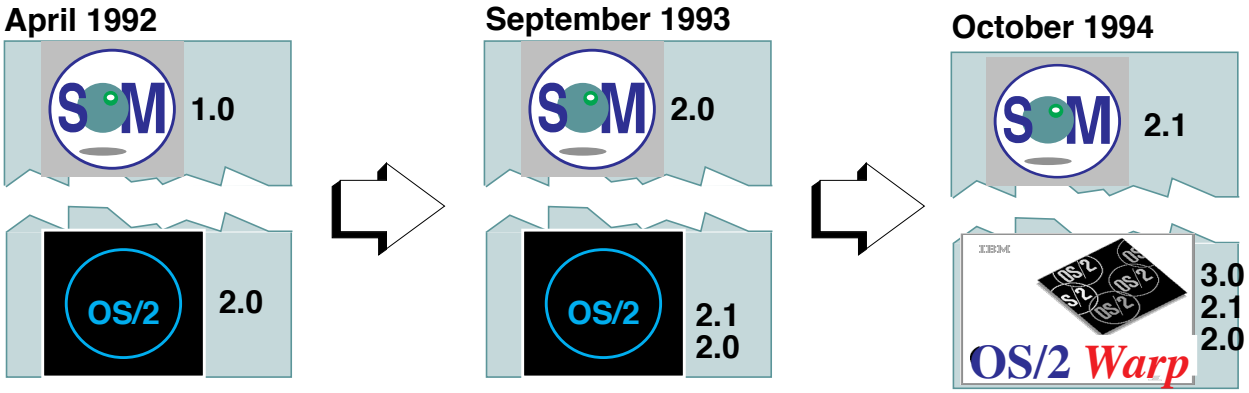

Figure 9. Evolution of SOMobjects Toolkit and the OS/2 Workplace Shell.

Raton, Florida) using the SOM 2.0 kernel, while the SOM team in Austin, Texas, completed the 2.1 effort. Only late in the Warp development (July, 1994) did the Workplace Shell development team receive the new SOM 2.1 kernel. Because of SOM's RRBC they were able to test without recompiling; allowing the SOM development team the independence of action necessary to fulfill its product requirements without impacting the schedule of the of the Workplace Shell development team.

This experience is only one of many similar experiences that might be cited as empirical proof of the efficacy of the SOM packaging technology. But the more fundamental point is that because of the RRBC emphasis in SOM such evolutionary development of independently shippable binary class libraries should be viewed as the norm.

Completeness of a Set of Transformations

A programming system is a programming language together with the tools that support it (e.g., compilers, linkers, and loaders). For any programming system, the definition of RRBC-successor (Definition 1) is sufficient to determine what ***are*** the safe transformations (transformations T such that for all libraries L_0, $T(L_0)$ is an RRBC-successor for L_0). However, that definition does not tell us what ***should be*** the safe transformations in any particular programming system. In this section, we propose such a definition based on the programming language alone. We argue that the current criterion for the correctness of separate compilation is too weak. This weakness has led to the dismal situation in C++ that Table 1 summarizes. A stronger criterion for the correctness of separate compilation is proposed.

Definition 1 is about binary compatibility, because each of the symbols A, L_0, and L_1 represents a compiled module. Let us be explicit about the compilation process; let the symbol τ represent the compilation process and let $P \oplus Q$ represent the joining of the source code of two modules P and Q. With these two symbols, we can represent the usual criterion for the correctness of separate compilation.

Weak Correctness Criterion for Separate Compilation

The separate compilation of a program A and a library L_0 is correct if

$SAT(\ A \oplus L_0,\ S\ )$ implies $SAT(\ \tau(A \oplus L_0^{def}) \bigcirc\!\!\!\!\bigcirc \tau(L_0),\ S\ )$

for all functional specifications S, where L_0^{def} represents the definitions that the language requires for the compilation of A.

For example, in C or C++, L_0^{def} represents the header files for the library L_0. On the other, for LISP libraries L_0^{def} is always an empty set of statements. The asymmetry in the formula is due to the expectation that the program knows about the library, but the library must contain no references to the program (in order for the library to be reusable). Again we have waved our hands at a number of important issues; particularly, the fact that in some languages separate compilation may require more work than merely separating the source code into different files. (The use of the symbols τ,$\oplus$, etc. is intended to clarify the presentation. We understand that we are not proving a mathematical result, and we hope you understand this too.)

The Weak Correctness Criterion is the origin of the release-to-release compatibility problem. If the library L_0 is changed, the application must be recompiled if the definitions change. As we have seen in the earlier sections of this paper, it is not sufficient for a programming system to support such a weak notion.

So let us rewrite Definition 1 using τ and letting A, L_0, and L_1 represent source code.

Definition 2. L_1 is a *RRBC-successor* to L_0 if

$SAT(\ \tau(A \oplus L_0^{def}) \bigcirc\!\!\!\!\bigcirc \tau(L_0),\ S\ )$ implies $SAT(\ \tau(A \oplus L_0^{def}) \bigcirc\!\!\!\!\bigcirc \tau(L_1),\ S\ )$

for all reasonable functional specifications S and for all applications A using library L.

This more precise definition explicitly shows that the application program is compiled with the definitions of the old library. Definition 2 is an improvement; however, the definition still relies on the notion of a reasonable functional specification. Let us define a *reasonable functional specification* as one that does not constrain or reference the structure of the program that implements the specification (that is, either the structure of the source program or the structure of the compiled program). This definition of reasonable functional specification has a deficiency in that reflective programs (programs that have outputs based on the structure of the program) do not have functional specifications that are reasonable. This is not a concern here; this definition of reasonable functional specification is adequate for defining what should be the

safe transformations and strengthening our notion of correctness of separate compilation.

Now let us assume that $L_1 = T(L_0)$ where T is a correctness preserving transformation on the library alone when the library and the application program are compiled as a single unit.

Definition 3. A transformation T (on a library L_0) is *correctness preserving* if
$SAT(A \oplus L_0, S)$ implies $SAT(A \oplus T(L_0), S)$
for all reasonable functional specifications S and for all programs $A \oplus L_0$.

We claim (and our experience supports) that a programming system is much easier to use when the correctness preserving transformations (on the library) are not invalidated by separate compilation. This is very natural to LISP programming (where L_0^{def} is empty) and C programming (where L_0^{def} is not empty). Thus, the definition of RRBC-safe transformations can based on that of correctness-preserving transformations.

Definition 4. If $SAT(A \oplus L_0, S)$, then a transformation T is *RRBC-safe* if
$SAT(A \oplus T(L_0), S)$ implies $SAT(\tau(A \oplus L_0^{def}) \bigcirc\!\!\!\!\bigcirc \tau(T(L_0)), S)$
for all reasonable functional specifications S and for all applications A using library L_0.

If Definition 4 is rearranged so that its central formula looks like this:

$[SAT(A \oplus L_0, S)$ implies $SAT(A \oplus T(L_0), S)]$ implies $SAT(\tau(A \oplus L_0^{def}) \bigcirc\!\!\!\!\bigcirc \tau(T(L_0)), S)$

then it is evident that an RRBC-safe transformation must be correctness-preserving. Note that transformations that change both the application program and the library may be correctness preserving, but are not RRBC-safe. Also, note that Transformation 2 is not considered RRBC-safe for C++ programs, because the antecedent of the definition is not fulfilled (if T changes the signature of a method, $A \oplus T(L_0)$ is not a valid C++ program).

Combining Definition 4 with the Weak Correctness Criterion for Separate Compilation yields a stronger criterion for the correctness of separate compilation for all programming systems.

Strong Correctness Criterion for Separate Compilation

The separate compilation of a program A and a library L_0 is correct if

$SAT(\ A \oplus L_0,\ S\ )$ implies

$[\ SAT(\ \tau(A \oplus L_0^{def})$ ⋈ $\tau(L_0),\ S\ )$

and

$SAT(\ A \oplus T(L_0),\ S\ )$ implies $SAT(\ \tau(A \oplus L_0^{def})$ ⋈ $\tau(T(L_0)),\ S\ )\ ]$

for all functional specifications S, where L_0^{def} represents the definitions that the language requires for the compilation of A.

This stronger criterion says:

> Any library transformation that preserves correctness when the library is compiled with a application program as a single unit should preserve correctness when the library and the application program (with the old definition context) are separately compiled.

Thus far, we have just restated our definitions in preparation for defining the completeness of a set of transformations for a programming system. A programming system is *RRBC-complete*, if all correctness-preserving transformations on libraries are RRBC-safe transformations. In procedural programming, a programming system that is not RRBC-complete would be deemed faulty and not be tolerated. For object-oriented programming, the same should be true. We further assert:

All programming systems should be RRBC-complete.

Conclusion

In all programming systems, the correctness preserving transformations on libraries should preserve Release-to-Release Binary Compatibility. Based on this, object-oriented programming need not give up implementation inheritance to achieve release-to-release binary compatibility. There are no fragile base classes, only programming systems that do not properly support subclassing across the binary interface.

Acknowledgments

We would like to thank Liane Acker, Arindam Banerji, Ravi Condamoor, Nissim Francez, Kevin Greene, Duane Hughes, John Irwin, Shmuel Katz, Vinoj Kumar, John Lamping, Simon Nash, Kim Rochat, Cun Xiao, and the OOPSLA reviewers for their comments. In addition, we thank Brad Cox, Ted Goldstein, and Andy Palay for their cooperation.

References

1. Apple Computer *Dylan: An object-oriented dynamic language* (1992).
2. Danforth, S., Koenen, P. and Tate, B. *Objects for OS/2* Van Nostrand Reinhold (1994).
3. Danforth, S.H. and I.R. Forman "Reflections on Metaclass Programming in SOM" *Proceedings of OOPSLA'94*, Portland, Oregon (October 23-26, 1994).
4. Forman, I.R., S.H. Danforth, and H.H. Madduri "Composition of Before/After Metaclasses in SOM" *Proceedings of OOPSLA'94*, Portland, Oregon (October 23-26, 1994).
5. Goldstein, T. C. and Sloane, A.D. "The Object Binary Interface -- C++ Objects for Evolvable Shared Class Libraries" *USENIX C++ Technical Conference* 1994.
6. Gosling, J., B. Joy, and G. Steele *The Java Language Specification* Addison-Wesley (to be published 1996 - chapter excerpts are available at http://www.javasoft.com/java.sun.com/newdocs.html#dev).
7. Graube, N. "Metaclass Compatibility" *OOPSLA '89 Conference Proceedings* (October 1-6, 1989) 305-316.
8. Hamilton, J., Klarer, R., Mendell, M., and Thomson, B. "Using SOM with C++" *C++ Report* (July/August 1995).
9. Kiczales, G., des Rivieres, J., and Bobrow, D. G. *The Art of the Metaobject Protocol* The MIT Press, Cambridge, Massachusetts (1991).
10. Object Management Group *The Common Object Request Broker: Architecture and Specification* OMG Document Number 91.12.1 Revision 1.1.
11. *OLE 2 Programmer's Reference Volume One* Microsoft Press (1994).
12. Palay, A.J. "C++ in a Changing Environment" *USENIX C++ Technical Conference* 1992.