9 References

[Bancilhon] F. Bancilhon, C Delobel, P. Kanellakis, *Building an Object Oriented Database System - The Story of O_2*. Morgan Kaufman, San Mateo CA, 1992.

[Bannerjee] J. Banerjee, W. Kim, H. Kim, H. Korth. “Semantics and Implementation of Schema Evolution in Object-Oriented Databases”, Proceedings of the ACM SIGMOD Conference, pages 311-322, 1987.

[Birrell & Nelson] A. D. Birrell & B. J. Nelson, “Implementing Remote Procedure Calls,” ACM Trans. Computer Systems, 2(1), February 1984

[Ellis & Stroustrup] M. Ellis & B. Stroustrup, *The Annotated C++ Reference Manual*, AW 1990.

[Goldstein] T. Goldstein, A. Sloane, M.Ball, A. Palay, “Supporting the Evolution of Class Definitions - Workshop Report” in OOPSLA’93 - Addendum to the Proceedings, to appear in SIGPLAN Notices, 1994.

[Hamilton & Radia] G. Hamilton and S. Radia, “Using Interface Inheritance to Address Problems in System Software Evolution”, Proceedings of the ACM Workshop on Interface Definition Languages, 1994.

[Harrison & Ossher] W. Harrison and H. Ossher “Extension by addition: Building extensible software.” Research Report RC16127, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 1990

[Lenkov] D. Lenkov, M. Mehta, S. Unni, “Type Identification in C++”, Proceedings of USENIX C++ Conference, Washington D.C. April 1991.

[OMG] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, August 1991, OMG Document Number 91.8.1.

[Palay] A. Palay, “C++ in a Changing Environment”, in Proceedings of the Usenix C++ Technical Conference, Portland, September 1992, pages 195-206.

[Penney & Stein] D. Penney & J. Stein, “Class modification in the GemStone object-oriented DBMS”, in Proceedings of the ACM OOPSLA Conference, pages 111-117, 1987.

[Richardson and Carey] J.E. Richardson and M.J. Carey, “Persistence in the E Language: issues and implementation,” *Software—Practice and Experience* number 19, pages 1115-1150, 1989

[SOM] *OS/2 2.0 Technical Library System Object Model Guide and Reference*, IBM Doc 10G6309.

[Stroustrup] B. Stroustrup, “Namespaces”, ANSI X3J16/93-0105, ISO WG21/N0312.

10 Acknowledgments

The authors gratefully acknowledge the help provided by Michael L. Powell of SunSoft and Michael S. Ball of SunPro, as well as all of the reviewers on the program committee. We would especially like to thank Brian T. Lewis for the thorough careful reading and editing of this paper.

Disadvantages of this approach are that it must be performed for *all* calls, and that it is difficult to recover after a failed check (possible actions are to call `abort` or raise an exception.)

(ii) "Migrate" the old `otbls` to new `otbls`.

In this case the entries for unsupported virtual functions are set to point to runtime error routines. This ultimately has the same error semantics as in option (i) , but there is no overhead of bounds checking for virtual functions. (In this way it is analogous to the use of thunks in a conventional `vtbl` to avoid unnecessary adjustments to the `this` pointer.)

(iii) Separate the notions of *compile-time type* and *run-time type*, and use the run-time `typeid` operator to give the programmer explicit control.

We observe that we wish the compile-time type system to treat all conformant versions of a type as the same (e.g. allow any version of the type `Bar` to be passed to a function declared as taking a parameter of type `Bar`), but that we may wish to distinguish between them at run-time. In other words applying the `typeid` operator to objects of different versions of a type would produce different results. The programmer could then explicitly test for version conformance and make appropriate decisions.

(iv) Reify the notion of namespace and allow it to dynamically correspond to a physical entity such as a dynamically linked shared library. For example create a construction which allows a namespace to be returned from a system level operation. `For example: namespace foo = dlopen(..);` Use the normal C++ type system to enforce conformance

At the present time we do not consider any of the approaches listed above to be satisfactory and we remain interested in exploring further approaches. Although (iv) is the most intriguing, we do not believe the C++ language can tolerate any additional major extensions.

The semantics we really require are those of the "newer is better" variety. A link time mechanism, such as that in SunOS 4.x, which supports a (major, minor) version pair to denote compatible versions provides the correct semantics. A change in major version denotes an incompatible change. A change in minor version denotes a compatible change, and all compatible changes are serialized i.e. there is no branching of compatible changes. The binary file resulting from the compilation and link-editing of a library or application records the version number of the dynamically linked library on which it depends. At run-time the dynamic linker selects according to its search rules a version of the library which is the "newest", which matches exactly in major version number, and for which the minor version number is greater than or equal to the minor version number specified in the client's dependency list. Thus there is only one version of a type present in the client's address space and it supports at least the functionality required by the client.

8 Conclusion

We have presented a new object model called the *Shared Object Model* and a new C++ implementation model called the *Object Binary Interface* (OBI. These models restrict some semantics of C++ programs to allow objects to span versions and to allow multiple implementations of an interface to co-exist within an address space. The shared object model supports virtual functions, public and protected data as well as derivation. The principal restriction over the normal C++ object model is that private data may only be accessed either explicitly or implicitly through the "this pointer." By using the linkage specification of C++ and allowing this key restriction of C++ usage, we are able to add an evolutionary version object model without the need for a language extension. Initial performance estimates indicate that this strategy is viable and adds little overhead to programs.

This is the OBI up level for both virtual and non-virtual inheritance: 4 instructions

```
ld    [%i0], %l1          ! get optr
ld    [%l1+A_IN_B_OFFSET],%l2 ! get the base class offset
add   %i0,%l2,%l4         ! add the base class offset to this
ld    [%l4],%l5           ! return the value
```

Table 5. Summary of Instruction Counts

	call up level non-virtual inheritance	call up level virtual inheritance	get an up level value non-virtual inheritance	get an up level value virtual inheritance
cfront	4	4	1	2
OBI	5	5	4	4

We believe that the overhead indicated is well within our acceptable levels. On modern multi-stage pipelined architecture, we believe that the actual execution time will be insignificant. Further study of the dynamic properties of the implementations are underway.

7 Object Version Migration

So long as the interface changes which resulted in the new version of a class are upward compatible, client code which was compiled against the old version works correctly when passed a "new" object. Although we recommend against exposing data members in software interfaces, it is still possible to have public and protected data members in this scheme. Public and protected data members are still present and are located at fixed offsets within the defining class. Our upward compatibility rules require that no public or protected members are reordered or deleted. Virtual functions retain the same index within the otbl which defines them (since upward compatible changes do not reorder or delete virtual functions).

A problem however arises when code compiled against the new version of the interface gets hold of an "old" object. How can this happen? This can arise if for example we have the following sequence of versions of an interface.

```
// version V1.0
class Bar { ... };
void Bar::f( Bar *pb ) { ... }

// version V1.1
class Bar { ... };
void Bar::f( Bar *pb ) { ... pb->g(); ... }
void Bar::g() { ... }          // new in version 1.1
```

If libraries implementing both versions of class `Bar`, V1.0 and V1.1, are linked into the program's address space, then "new" code (in V1.1) can be passed an "old" object (created by V1.0) and attempt to invoke an operation - such as `g()` - on it which the V1.0 object does not support.

At the present time we are exploring possible solutions to the version migration problem. Note that in the case where shared linkage is used only to support evolution, and only one version of a library is linked to a program at one time, there are no such "version migration" problems. Among the approaches we have considered are:

(i) Perform a runtime bounds check on the virtual function invocation, testing the `otbl` index against the value of `n_functions`. A similar check is necessary for access to public and protected data members.

```
struct __smfptr {
      size_t base_index;       // index of base
                               // -1 if direct
      size_t index;            // index of virtual
                               // negative if non-virtual
      fptr  faddr;             // only for non-virtual
};
```

Note that the offset of the `optr` need not be stored, since it is always at offset zero. The base index is used to calculate the adjustment to be applied to the `this` pointer, by a lookup in the `otbl`.

6 Instruction Counts

Most operations are identical to cfront. This includes getting data and calling virtual functions at the level of the type the data and function is defined at. The only differences occur when accessing data or calling a function in a base class. For the cfront implementation model, the are different code sequences for virtual and non-virtual inheritance. For the OBI implementation model, the code sequences are the same for virtual and non-virtual inheritance. Table 5 summarizes the instruction count difference.

6.1 Calling a virtual function defined in a base class.

This is case for current cfront non-virtual inheritance: 4 instructions

```
ld    [%i0+8],%l1             ! load the vptr into l1
ldd   [%l1+8],%l2,%l3         ! load double the thisDelta
                              ! and function addr into l2,l3
add   %l2,%o0,%o0             ! adjust the thisDelta
call  %l3
```

This is the case for current cfront virtual inheritance from a virtual base: 4 instructions

```
ld    [%i0+OFFSET_PB],%o0     ! load pB->PB into o0
ld    [%o0+8],%l1             ! load the vptr into l1
ld    [%i1+12],%l2            ! load the function addr into l2
                              ! the adjusted this pointer is
                              ! already in o0)
call  %l2
```

This is the case for OBI both virtual and non-virtual inheritance call up level: 5 instructions

```
ld    [%i0],%l1               ! load the optr into l1
ld    [%l1+A_IN_B_OFFSET],%l3 ! load B's A base into l3
ldd   [%l3+8],%l4,%l5         ! load double the thisDelta and
                              ! function addr into l4,l5
add   %i0,%l5,%o0             ! add the thisDelta to this
call  %l4
```

6.2 Accessing a member in a base class

Current cfront non virtual inheritance: 1 instructions

```
ld    [%i0+OFFSET_OF_MEMBER],%l1
```

This is the current cfront virtual inheritance: 2 instructions

```
ld    [%i0+OFFSET_PB],%l0          ! load B's A base into %l0
ld    [%l0+OFFSET_OF_MEMBER],%l1   ! indirect through PA
```

FIGURE 9. Virtual function call in shared linkage.

```
class A {
public:
      virtual void vfA1();
      virtual void vfA2();
};

class B : public A {
public:
      void vfA1();      // override
      virtual void vfB();
};

B* pB = new B;

pB->vfB();              // (1)
pB->vfA1();             // (2)
pB->vfA2();             // (3)
```

This is implemented approximately as follows. Note that we show the implicit `this` parameter for purposes of illustration only.

```
__otbl* ot = pB->optr;

function_part* fp - &(ot->function[1]);                    // (1)
fp->*fct(this + fp->this_adj);

fp = &(ot->function[0]);                                   // (2)
fp->*fct(this + fp->this_adj);

ot = ot->base[0].base;                                     // (3)
fp = &(ot->function[1]);
fp->*fct(this + fp->this_adj);
```

Note that in the third case the client code references the *nested otbl* for class A in class B (nested tables are built even in the case of single inheritance). Thus in later versions derived classes can add overrides of base class virtual functions.

5.13 Overriding the return type of virtual functions

To implement calls of virtuals in which the return type has been overridden we use the same mechanism as for ordinary virtual function calls with the additional step of adding the return adjustment stored in `ret_adj`. Note that in fact *both* adjustments must always be done, since at run-time there is no knowledge of whether or not the function return type has been overridden.

This implementation does not support cases where the overriding contraviant return type is a class with shared linkage or is virtually derived from the original type. A full implementation requires the execution of a complex code sequence that calculates the return adjustment based on the actual type of the object.

5.14 Pointer to Member-Function

The pointer-to-member-function machismo is similar to the mechanism described in Section 5.5 on page 10. Pointers to member functions are represented by a structure `__smfptr`, defined as follows

FIGURE 8. Algorithm for the initialization of otbls

```
__otbl* MY_TYPE::otbl_creator_function()
{
__otbl *my_otbl ← allocate(sizeof(fixed_part) +
            n_bases * sizeof_(base_part) +
            n_functions * sizeof(function_part));
my_otbl->n_functions ← MY_TYPE_NUMBER_OF_FUNCTIONS;
my_otbl->n_bases ← MY_TYPE_NUMBER_OF_BASES;
my_otbl->size ← sizeof(MY_DATA) + sizeof(_otbl*);
my_otbl->most_widen_offset = 0;

//Size and assign position of all base classes
foreach direct base otbl_b of MY_TYPE in traversal order do
   {
   my_otbl->base_b ← call TYPE_b::otbl_creator_function();
   // unify previous allocated virtual base class
   foreach base otbl_bb of otbl_b
      {
      is otbl_bb already allocated virtually?
         {
            delete base_bb;
            base_bb ← previously allocated otbl;
         }
      else // Not seen yet. Allocate its region .
         {
            base_bb->most_widen_offset = - my_otbl->size;
            my_otbl->size += base_bb->size ;
         }
      }
   }
// Assign the values for the virtual function part
foreach base otbl_bbb both direct and indirect
   {
   foreach function_f in base_bbb
      {
      if MY_TYPE overrides function_f
         {
         base_bbb->function_f->fct ← function;
         base_bbb->function_f->this_adj ← 0;
         }
      else
         base_bbb->function_f->this_adj ← base_bbb.widen_offset;
      if function_f has a covariant return type == MY_TYPE
            base_bbb->function_f->ret_adj ← COVARIANT_TYPE_OFFSET_f ;
      }
   }
return my_otbl;
}
```

5.12 Virtual Function Calls

Calling a virtual function for a class with shared linkage involves an indirection through one or two `otbls`, depending on whether the function is defined in the derived class or inherited from a base class. Figure 9 below illustrates the implementation.

```
extern "shared" class A { public:int a1, a2;};
class B {
public: int b1;
      A ba;
      int b2;
} myB;
      myB.b2 = 999;
```

The offset of the member `b2` of `B` is dependent on the size of `class A`, which must be looked up in the `otbl`. So the code to make the assignment to `myB.b2` looks like

```
* (int*)((char*)&myB + OFFSETOF(B::ba) + myB.optr->size)) = 999;
```

5.8 Static Data Members

Static data members are unaffected by a "`shared`" linkage specification.

5.9 Static Class Functions and friend functions

For static class functions and friend functions, there is no access to the private data members. This is because these functions do not vector through the virtual function table. There is thus no way to ensure that the compile-time correspondence of the private data of the `this` pointer actually corresponds to the version of the implementation that is reached by these functions.

5.10 Constructors, operator new and sizeof

As in default linkage `operator new` is always called before the constructor. Note that allocating an object of a class with shared linkage always requires a call to `sizeof` no matter what the storage class the user specifies. Moreover in shared linkage `sizeof` is not a compile-time operator, but requires a lookup of the `otbl`. We can illustrate its implementation with pseudo-code as follows

```
size_t sizeof(TYPE) { return TYPE_otbl.size; }
```

But since there might be more than one implementation of type, it is more interesting to use `sizeof` on an instance.

```
size_t sizeof(void* inst)
{
      void* derived = ((char *)inst) + inst->_optr->most_widen_offset;
      __otbl* ot_of_derived = *((__otbl**) &derived);
      return ot_of_derived->size;
}
```

5.11 Initialization of Otbls

Since values in the `otbl` representing instance size, base object offsets, and virtual functions are not known until run-time (when the dynamic linker is run at program initialization time, or after a `dlopen`) the `otbl` cannot be initialized statically. However it must be initialized before any instance of the class can be allocated. In particular, the `otbl` must be allocated before any call to `sizeof`. Thus the `otbl` can be initialized by `.init` code in the shared object, or by code run dynamically the first time `sizeof` is called for the type and the result cached for further use.

The initialization of the `otbl` is done by calling a built-in static member function for each class with shared linkage, `otbl_creator_function`. Figure 9 below describes the otbl creation algorithm.

Unlike normal vtbls, there is no difference in the code generated for casts to virtual and non-virtual bases[1].In the example,

```
__otbl* ot = pC->optr;
```

sets `ot` pointing to `C_otbl`. Then

```
ot->base[1].offset // has the value 16.
```

Adding `16` to `pC` sets `pB` pointing to the object representing the `B` part of `C`, and the `optr` for that object points to `BinC_otbl`.

The cast to `V*` is implemented as follows

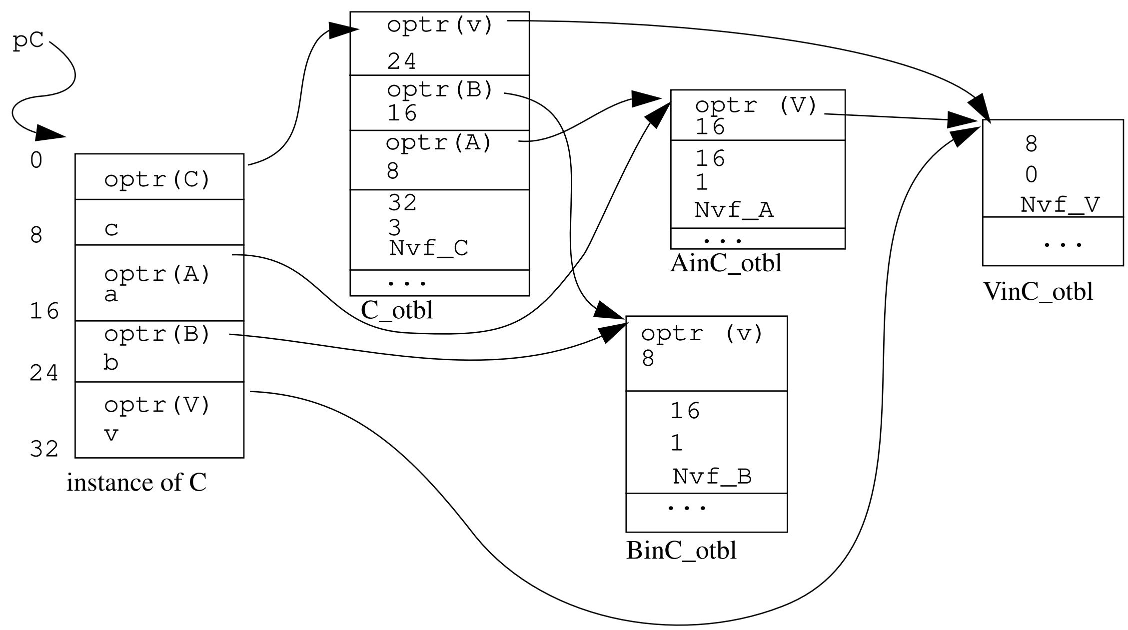
```
ot = pB->optr;
```

sets `ot` pointing to `BinC_otbl`.

```
ot->base[0].offset // has the value 8.
```

Finally, adding `8` to `pB` sets `pV` pointing to the object representing the `V` part of `B` (and of `A` and `C` since it's virtual), and the `optr` for that object points to `VinC_otbl`.

FIGURE 7. Layout of derived class instance and otbls



5.7 Composition

By *composition* we mean the declaration of a member of a class which is itself of a class type. This is the common use o embedding one class in another as in:

1. The GNU C++ compiler, `g++`, uses a similar scheme to implement virtual derivation in general where the virtual base pointers are replaced by offsets stored in the `vtbl`.

5.6 Derivation and Casts

Casts are implemented much as in conventional implementations, see for example [Ellis & Stroustrup] pages 221-227, except that the adjustments applied to pointers are not compile-time constants, but rather must be looked up in the object's `otbl`.

Downcasts (i.e. casts from base to derived class) are implemented by casting to the most derived class using the `most_widen` field, and then back to the intermediate class. As a consequence of this, downcasts from virtual bases are supported, but the result of downcasting to a non-virtual intermediate base which occurs twice in the hierarchy is undefined. This is the only difference we have identified between shared linkage and "standard" C++, and many users find it less restrictive than the absence of casts from virtual bases. In passing we note that this same restriction is also present in SOM's C++ mapping.

FIGURE 6. Cast from a derived class to a base class

```
extern "shared" {
      class V { public: int v; private: ... };
      class A : public virtual V {
            ...
      public:
            int a:
      private:
            ...
      };
      class B : public virtual V {
      public:
            int b;
      private:
            ...
      };
      class C : public A, public B {
      public:
            int c;
      private:
            ...
      };
}

C* pC = new C;
B* pB = pC;// implicit cast, B* ⟵ C*
V* pV = pB; // implicit cast, V* ⟵ B*
```

The layout of the object pointed to by `pC`, and it's network of `otbls` is shown in Figure 9 below. Pseudo-code illustrating the implementation of the casts is

```
__otbl* ot = pC->optr;
pB = (B*) ((char*)pC + ot->base[1].offset);
__otbl* ot2 = pB->optr;// B in C otbl
pV = (V*) ((char*)pB + ot2->base[0].offset)
```

protected members, since they are laid out in declaration order immediately after the `optr` and before the private members. We require that implementation code for the class, i.e. member function and friend function bodies, which directly accesses the private members must always be compiled against the up-to-date version of the class definition, and so the compile-time determined offsets are valid. To access a member defined in a base class[1] we proceed in the following steps

(i) follow the object's `optr` to its `otbl`

(ii) index in the `base` array to find the appropriate `base_part` structure

(iii) add the `widen_offset` to the address of the object to find the base class object

(iv) add the offset of the member within the base class.

Thus the C++ statement "`myB.a2= 999;`" is equivalent to the following C statement

```
((A*)((char*)&myB + myB.optr->base[0].offset))->a2= 999;
```

5.5 Pointers to Data Members

A pointer to data member is represented by an instance of class `__smdptr`, which is defined as follows:

```
struct __smdptr {
      size_t base_index;
      size_t offset;
};
```

The `base_index` is negative if the it a direct member, and not a member of a base class. An example of accessing a pointer-to-data-member is:

```
extern "shared" class B {
public:
      int b1:
      int b2;
};

extern "shared" class D : public B {
public:
      int d:
};

int D::* p = &D::b2;
D* pD = new D;
pD->*p = 999;
```

The pointer `p` is represented as

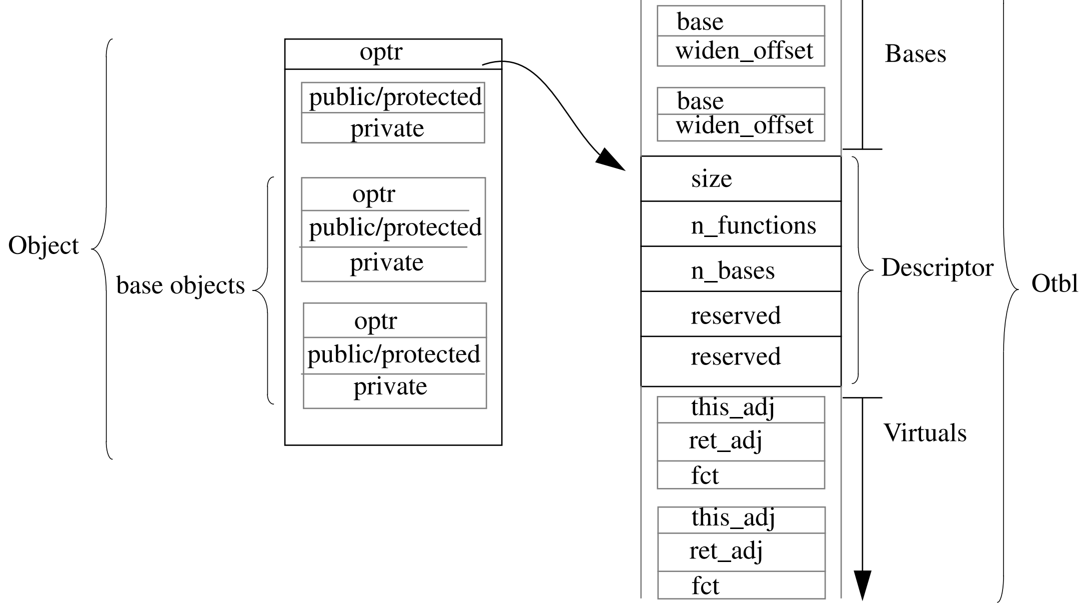
```
__smdptr p = {1,4};
```

Referencing through `p` results in a lookup of the appropriate `base_part` structure in the `otbl`

```
__otbl* ot = pD->optr;
char* pTmp =(B*)((char*)pD + (p.base_index < 0 ? 0 :
                  ot->base[p.base_index].widen_offset));
* ( (int*) ( pTmp + (char*)p.offset ) ) = 999;
```

1. In client code this must be a public or protected member of a public or protected base

FIGURE 4. Object and Otbl



5.3 Allocation of automatic and static instances

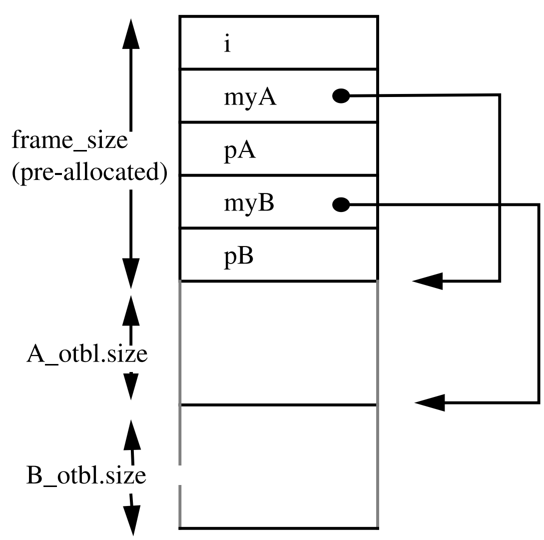
Automatic objects of a class with shared linkage are allocated on the stack frame via an `alloca`-like mechanism detailed below. Objects are represented by a pointer to a location elsewhere in the stack frame and the "real" object is laid out by special prologue code in the function. Static objects, whether global or file static, are represented by a pointer and the "real" object is on the heap since the real object is not known. Allocation and initialization of statics are done (as is conventional) with `.init` code.

FIGURE 5. Allocation of auto objects in Shared Linkage

```
extern "shared" {
      class A {
      public:
            int a1;
            int a2;
      };

      class B : public A {
      public:
            int b;
      };
}

void func() {
      int i;
      A myA, *pA;
      B myB, *pB;
}
```



5.4 Access to Data Members

Just as in default linkage, to access a member defined in the class itself (not a base class) we use an offset determined at compile-time. Notice that changing the size of the private part or extending the list of public and protected members does not change the offsets compiled into client code for existing public and

As with many simple ideas, these two rules generate many important implementation details. The rest of this paper describes the OBI implementation model, including several key algorithms.

5.1 Otbls

For classes with shared linkage, `otbls` take the place of `vtbls`. The terms `optr` and `otbl` originate from the `vptr` and `vtbl` described in [Ellis & Stroustrup]. The 'o' may stand for 'offset' or 'object'. Within an object, the public and protected data members immediately follow the `optr`. The object's `optr` points to the `size` field in the fixed part of the `otbl`. The `base_part` structures are indexed backwards from there, and `function_part` structures are indexed forwards. The `base_part` structures are indexed in order on a left-to-right, pre-order depth-first traversal of the inheritance DAG, with `base_part`'s for virtual bases left to the end and allocated in the order the virtual bases were encountered in the graph traversal. The `function_part` structures are allocated for each virtual function defined in the class in order of declaration.

An `otbl` is global data, and will have an external "mangled" name. It can be treated as a global variable or (probably better) as a private static data member of its class. One difference between `vtbls` and `otbls`, is that `vtbls` along the left linear tree walk of the derivation are concatenated. This concatenation is not possible with `otbls`, as it would prohibit growth in adding new virtual functions to the end.

5.2 Class Layout

Within the public/protected section and the private section the members are laid out in declaration order. The order in which the base classes are laid out is unimportant, since all accesses to inherited members are calculated through offsets found in the `otbl`. The `otbl` consists of three parts:

(i) an array of `base_part` structures which grows backwards providing base class information. Each `base_part` consists of a pointer to an `otbl` and the offset of the base class object

```
struct __otbl{
     struct base_part {
        __otbl* base;
        size_t widen_offset;
     } base[n_bases];
```

(ii) a fixed descriptor section describing the size of instances, the number of direct virtual functions, and the total number of base classes. This part also contains access to the runtime type information.

```
     size_t size;
     size_t n_functions;
     size_t n_bases;
     TypeInfo* tinfo;
     size_t most_widen_offset;
```

(iii) an array of `function_part` structures which consist of a pointer to a direct virtual function of the class and adjustments for the object pointer and the return value.

```
struct function_part {
        size_t this_adj;
        size_t ret_adj;
        void (*fct)();
     } function[n_functions];
```

The relationship between an object and its otbl is presented in Figure 4 on the next page. The otbl points to the "otbl descriptor" which is located in the middle of the object. In the following sections we show through pseudo-code and diagrams how classes with shared linkage and objects of such classes are used. This description covers *all* possible uses of the object by client code.

```
extern "C++" {
      void f(int);
      class A { ... };
      class B : public A { ... };
};
class C { ... };
extern "shared" {
      int g(X*);
      class X { ... };
      class Y : public X { ... };
};
```

The function `void f(int)` and the classes `A`, `B` and `C` have default or "`C++`" linkage; and the function g(X*) and the classes `X`, and `Y` have "`shared`" linkage. A given class can have only one linkage specification throughout a program (This is the same rule which applies to the linkage specification of functions in the C++ draft standard). Moreover the representation of a *pointer-to-member-of-shared-class* is different to the representation of a *pointer-to-member-of-default-class* and assignment of one to the other is not allowed. In addition there is a semantic restriction on how classes with different linkage specifications can be combined. A derived class and all its base classes must have the same linkage specification.

4.2 Semantics of the Shared Object Model

The principle new rule of the OBI is that the private data of an object must be strictly encapsulated. Thus the 1990's corollary of "only friends may touch your private parts is that "*no one else may touch your private parts*." The official rules of the OBI are:

(i) The only access to the private data members of an object is through the object's (explicit or implicit) `this` pointer. Non-virtual functions, both member functions and non-member friend functions, cannot access the private parts. The shared object model does not prohibit access to public or protected data members.

(ii) New public and protected members (both data-members and virtual functions) must be added to the end of the list of existing members.

(iii) The one definition rule for object types is relaxed for shared linkage types within a program (but not within a single compilation unit).

5 The OBI implementation model

The shared object model is implemented using the Object Binary Interface (OBI) implementation model. Of course, users do not need to know the mechanics of the OBI implementation model. The principle notion is that the OBI is a binary interface similar in spirit to the System V Application Binary Interface (ABI). It is possible that a future generation of the OBI may allow for binary compatible linkage of other Object-oriented languages besides C++. The OBI has three basic concepts:

(i) The OBI specifies that each instance of a class with shared linkage contains an `optr` (pronounced oh-pointer) as its first element. The `optr` points to an `otbl` which describes the layout of the class instance and how to invoke the virtual functions of the class.

(ii) Public and protected data members come first. After all public and protected data members, come the private data members. Consequently the public and protected data members of a class can be extended because only the private data is adjusted. But the total ordering of the public and protected data members is preserved!

(iii) All inheritance is implemented in a fashion similar to virtual inheritance. The fundamental mechanism is to allow the relationship between base and derived objects to change and evolve. The virtual inheritance mechanisms in `cfront` provide the inspiration for the necessary indirection.

4 The Shared Object Model

The Shared Object Model provides a C++ programmer with an additional annotation to partition class library into classes that the implementor guarantees never to change, and classes that may change. The first category often corresponds to “key” components of the library whose interface is extremely well understood and for which optimal implementations are well known. The overall run-time performance of the library is dominated by the performance of these key components. For example, an implementor of a reference counting object might decide to exposure via inline functions the implementation. The implementor is trading off the risk of the need to change the implementation of the reference counting object in order to gain better performance. Inlines effectively exports the implementation (e.g. the layout of the private members) of a class and so precludes any possibility of evolution.

The second category corresponds to outer layers of abstraction, usually having more complex semantics and larger granularity. Thus adding overhead to function calls for these layers will have less impact on the performance of the library as a whole. Correspondingly these are the layers whose semantics are most subject to change from version to version and where there is greater variability in implementation. They are the classes where support for evolution is most important. Support for evolution is not appropriate for all classes, but where library design makes a natural partitioning of classes such that evolution is both possible and effective for critical classes.

Recognizing this, we chose to add extra syntax to identify evolvable classes to the compiler. We considered using a `#pragma`, but instead chose to use a linkage specification: `extern "shared" { ... }`. A linkage specification fits very well with our requirements:

(i) It is not a language extension. We abhor any more extensions to this language. But the C++ draft standard allows an implementation to provide an arbitrary set of linkage specifications in addition to the required specifications `"C"` and `"C++"`. We have defined a specification `"shared"`, which results in linkage conventions suitable for long-lived interfaces and for linkage across shared object boundaries.

(ii) Code using linkage specifications is portable. Compilers which do not support a particular linkage specification will generate a warning, but supply default linkage.

(iii) It is semantically accurate. The concept of “linkage” is concerned with communication across translation unit and library boundaries, especially if it provides cross-language models.

(iv) It is syntactically appropriate. Entire header files can be easily “wrapped” to use shared linkage, just as existing C header files could be given C linkage with `extern "C"`.

4.1 extern “shared” { ... }

We chose the word *shared* because we wanted to allude to System V shared libraries (dynamically linked libraries), and because most of the suggested alternatives, such as `dynamic`, `global` and `export`, had other conflicting connotations. In our implementation a “`shared`” linkage specification affects only class definitions and their member functions and does not affect non-member functions. The layout of class objects and the mechanism for calling virtual functions depends on the *linkage specification* of the class. For example in the following code fragment:

2.3 Schema Evolution

The object-database community has long recognized the necessity for supporting type and schema evolution. Object-database systems that support evolution include: Orion [Bannerjee], GemStone [Penney & Stein] and O2 [Bancilhon]. The E compiler in the Exodus system [Richardson and Carey] follows a technique similar to the OBI in using offsets to point to virtual base classes. But no support for multiple versions of an object class in a database has been described in this work.

2.4 Design focus of the OBI

All the mechanisms that have been proposed to support evolution of class definitions either involve significant extra indirection at run-time or require extra relocation work by the runtime linker. Linker start-up latency is already noticeable in dynimcally-linked C++ programs. Our concern is that additional start-up latency would be unacceptable.

Unlike ΔC++ and SOM we believe that not all changes of a class interface are of equal utility between version release. During software development, radical rearrangement of function order and other class hierarchy is useful. There is little value in supporting arbitrary reordering of member functions between two *released* versions of a class. Our selections of changes and usage rules provide compatible evolution of types and interface-implementation independence with limited impact on C++ usage. The OBI design highest priority is to allow the private parts of an object to change arbitrarily. The second highest priority is to allow extension by addition [Harrison & Ossher]. Other changes such as rearrangement of class hierarchies are not unimportant, but are not the focus of this work. If desired, it is simple to add a #pragma annotation which specifies the order of data members. The OBI design differs from previous work in that it focuses on a select few high values changes and it supports multiple simultaneous implementations of a class within an address space.

3 Existing Support of versioning in C++

There are several techniques that might support versioning within C++ including renaming, derivation, and namespaces. All of these techniques provide some handle on the problem, but all are insufficient to meet our criteria stated above. Renaming functions, for example has been around as long there have been symbolic identifiers. Version 6 UNIX had a file seeking operating system call named `seek` which took one 16 bit integer argument. The obvious flaw of files exceeding 64 K was fixed in version 7 UNIX by adding in the operating system call named `lseek`. This works amazingly well because both the old and new operations can coexist. In practice there is no limit to the number of “l”’s or other version-specific characters one can add onto an identifier. But exceedingly long identifiers become unwieldy and are a blight on the code.

Objects require additional support. Users would like new objects to be accepted anywhere an old object was accepted. One solution, proposed in [Hamilton & Radia], is to represent version conformance explicitly by modeling it using derivation.

```
class UNIX_File_v6 {
      unsigned short seek(unsigned short offset, int whence);
};
class UNIX_File_v7 : public UNIX_File_v6 {
      unsigned long seek(unsigned long offset, int whence);
};
```

As an alternative the recently defined namespace extension [Stroustrup] to the ANSI/ISO C++ draft standard provides a mechanism that can alleviate the unwieldiness of the long identifiers. Applications that contain objects derived from the UNIX_File_v6 class must be edited and recompiled to take advantage of the newer UNIX_File_v7 class. We want a solution that allows application objects to transparently use the latest version of the class. The approach we have chosen makes versions orthogonal to the C++ type system.

version differences between XLib R15 and XLib R16 created objects. The code mechanisms of unmarshalling (and dynamic loading) arrange to get the correct addresses for the new library into the virtual function tables.

A single CADMaker application will contain instances derived from both R15 and R16 XLib code. Enabling such multiple versions can be easily achieved by strictly respecting the encapsulation boundary of objects. An object should not access the private data of any other object, even those of its own class, except through virtual function calls. The private data of an object is truly kept private. Thus the only restriction over the normal C++ object model is that private data may only be accessed either explicitly or implicitly through the "this pointer."

These usage rules place restrictions on what the library vendor can do with C++ objects. Despite these restrictions, the advantages of allowing multiple definitions of an object in the same address space at the same time is significant and worthwhile for many applications. Note also that even in ordinary non-distributed applications dynamic loading makes it possible to have multiple implementations of an object coexist in the same address space.We will refer to the property of supporting multiple implementations of a type within the one address space as *interface-implementation independence*.

2 Related work

Several previous authors have discussed the issue of versions and evolution, some in the context of programming languages, but most in relation to schema evolution in object-oriented databases. A brief summary of these approaches is described in [Goldstein].

2.1 ΔC++

ΔC++'s solution does not support multiple implementations of an object in an address simultaneously. ΔC++ allows a developer to make compatible changes to a class with minimal recompilation of clients. The set of compatible changes maintains or extends the class interface without altering the code sequence used to access members of that class. ΔC++ accomplishes this by resolving classes at link time. Changes are handled by extending the linker to support new relocation types such as member offset. Most code sequences are identical to `cfront` [Ellis & Stroustrup] generated code except for constructors/destructors, calls to non-virtual members, access to embedded structures and some optimizations. The expense of this technique is additional program start-up time latency while the linker processes the additional relocations.

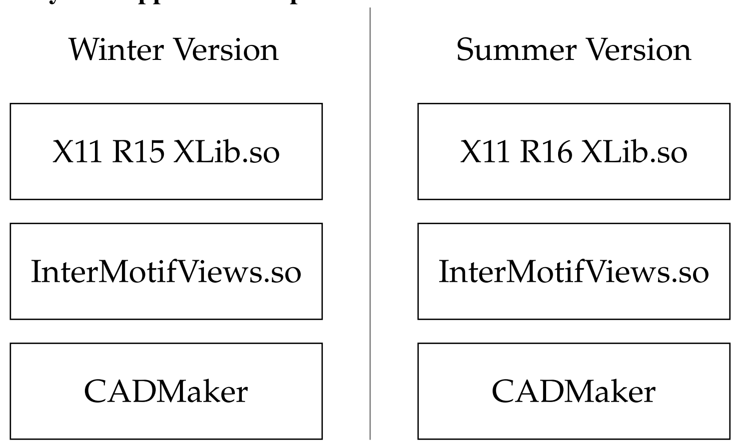
2.2 System Object Model

The System Object Model (SOM) was developed at IBM. There are several variants - SOM, DSOM that supports distributed objects within the general framework of the Object Management Group's CORBA, and PSOM that supports persistent objects. All of these provide run-time support for a common interface to "objects" independent of programming language. They also have an extensive metaclass framework, which in the DSOM case supports the dynamic invocation interface of CORBA.

By extending IDL's notion of interfaces, [OMG], with an implementation construct called "release-order:" SOM supports extension and change of an interface. Other aspects of class evolution are implemented by SOM's metaclass protocol. SOM puts the burden on the user to maintain the correct release order across versions. For example, the release-order entries must only be extended, never removed or deleted. SOM uses a dispatch mechanism that supports multiple inheritance but favors single inheritance implementations. SOM is a hybrid of the Smalltalk metaclass object protocol and the OMG object model layered on top of C++. Additional overhead is required for multiple inheritance involving caching and hash lookup tables. The principal difference between the OBI and SOM is that the OBI approach still assumes the essential mechanism and efficiency found in C++, while SOM emulates the object model using Smalltalk-like framework.

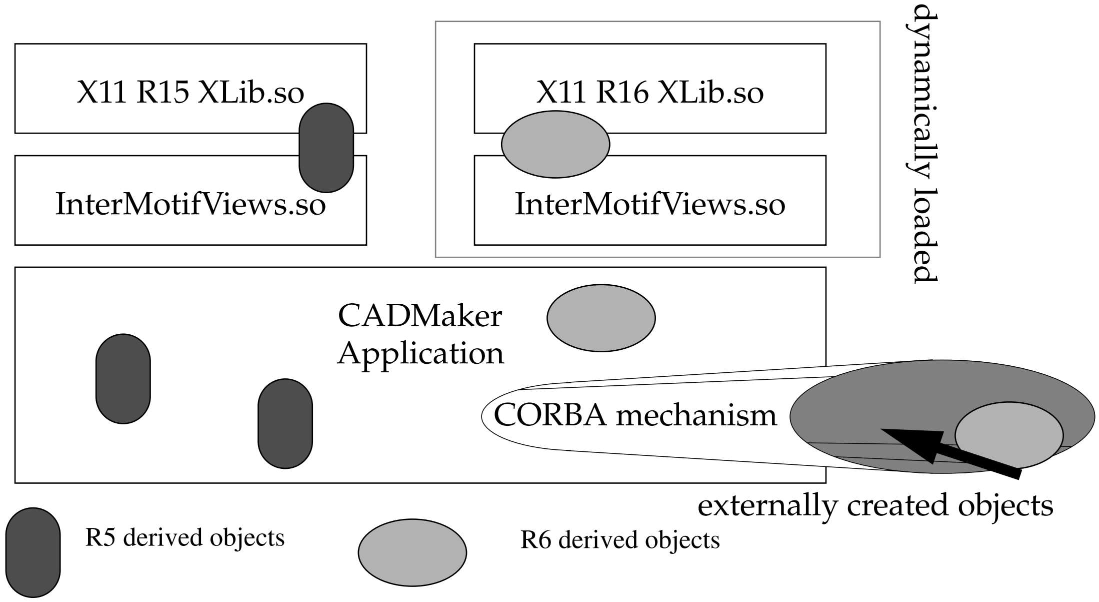
applications using InterMotifViews is a sophisticated network oriented application called *CADMaker*. Since XLib and InterMotifViews come from separate vendors, their will always be a time delay between release of the XLib library and the next release of InterMotifViews. It may even be possible that end-users receive updates of the X11 library before the maker of InterMotifViews sees it. Figure 2 depicts a Winter and following Summer rconfiguration of the CADMaker application.

FIGURE 2. Library and Application Dependencies



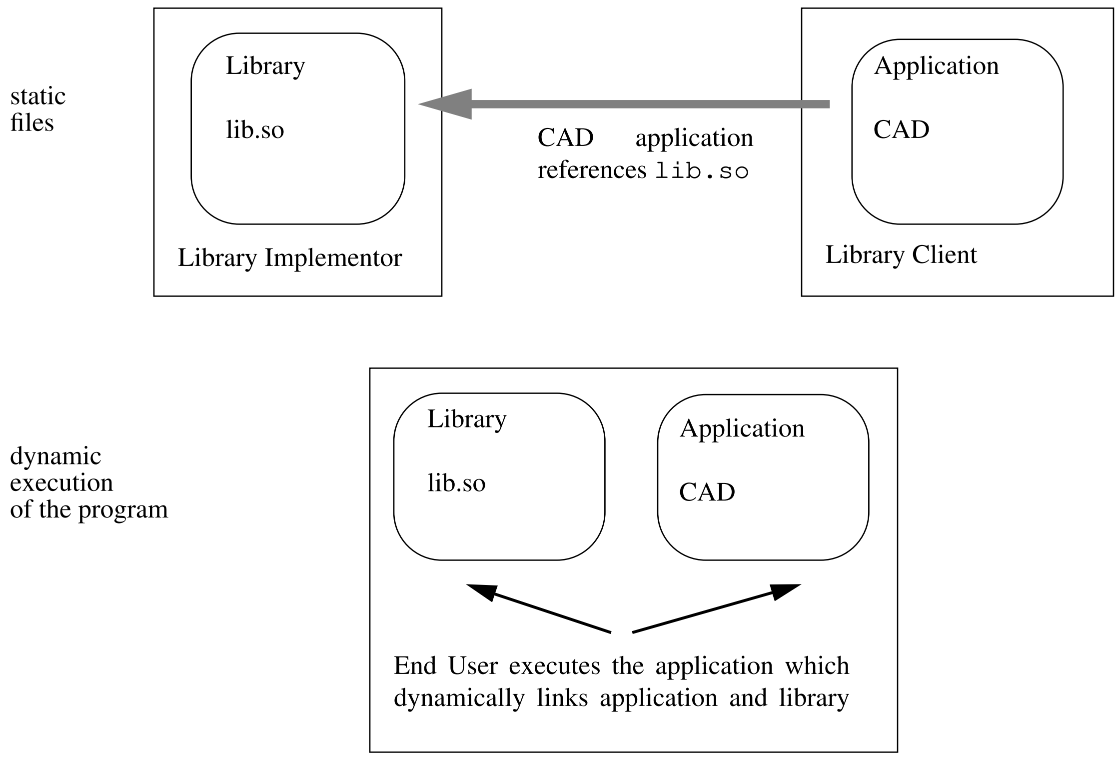
Suppose the environment and libraries supports multiple user distributed applications. The hypothetical InterMotifViews is designed for running workgroup applications which allow users to work together. As well, some servers at a CADMaker client site are running the Winter version and some are running the Summer version of XLib. CADmaker programs share objects. Objects which are made by compatible but different class libraries in an external program are transferred and shared from one address space to another through mechanisms such as CORBA. Figure 3 depicts the summertime execution context of CADMaker. where X11 R16 implementation objects come from an external address space marshalled through some RPC mechanism.

FIGURE 3. Marshalled objects



The correct behavior is that the application has access to both R15 and R16 libraries simultaneously. If XLib objects are accessed only through virtual functions, the CADMaker application should be unaware of the

FIGURE 1. Library, Application and End-User



Suppose the library implementor changes the implementation of `lib.so` or changes the interface in an upwardly compatible way to provide additional functionality and ships the new version to the end-user. The end-user expects - quite reasonably - that the expensive CAD package bought from the ISV will still run. The end-user cannot recompile the CAD package - she doesn't have sources, and may not even have a compiler. On the other hand the library-implementor, even if it's the platform library implementor, can't require all ISV's to synchronize with them. In reality, synchronization will be even more complicated since the CAD package likely requires libraries from several different vendors. To fulfill the promise of modular software components, object-oriented technology must support compatible replacement of software libraries.

It must be possible to introduce compatible changes to a library without requiring any changes to the source code or even recompilation of the clients of the library. Run-time mechanisms for object-oriented languages must support *compatible evolution* of class libraries.

1.2 Interface-Implementation Independence

Recent technologies such as the Object Management Group's CORBA [OMG] which support distributed applications allow objects to span address spaces. Objects may be defined in one program and used in another. Objects move from one address space to another through a linearization and communication process such as *remote procedure call* [Birrell & Nelson] or Object Invocation [OMG]. A very useful mechanism is for the external address space to pass an instance of a derived class. Frequently, during the unmarshalling of the object, it may be discovered that the new address space does not have the derived class implementation of the object, but only have the base class interface of the object. Dynamic linking provides a mechanism a program may use to acquire the corresponding implementation for a marshalled object.

Imagine, for example, that X11's *XLib* library is written in C++. Suppose there is a hypothetical third party library vendor writing a window system toolkit called *InterMotifViews* that uses XLib. Among the

The Object Binary Interface — C++ Objects for Evolvable Shared Class Libraries

Theodore C. Goldstein
Sun Microsystems Laboratories
ted.goldstein@eng.sun.com

Alan D. Sloane
SunPro
alan.sloane@eng.sun.com

Abstract

Object-oriented design and object-oriented languages support the development of independent software components such as class libraries. When using such components, versioning becomes a key issue. While various ad-hoc techniques and coding idioms have been used to provide versioning, all of these techniques have deficiencies - ambiguity, the necessity of recompilation or re-coding, or the loss of binary compatibility of programs. Components from different software vendors are versioned at different times. Maintaining compatibility between versions must be consciously engineered. New technologies such as distributed objects further complicate libraries by requiring multiple implementations of a type simultaneously in a program.

This paper describes a new C++ object model called the *Shared Object Model* for C++ users and a new implementation model called the *Object Binary Interface* for C++ implementors.These techniques provide a mechanism for allowing multiple implementations of an object in a program. Early analysis of this approach has shown it to have performance broadly comparable to conventional implementations.

1 Introduction

Software either evolves or dies. Software evolution occurs in response to numerous requirements including bug fixes, user demands for greater functionality, and especially to support changes in related software. Object-oriented programming promises to allow individual class library components to evolve independently of its clients. Many modern operating systems allow software libraries to be efficiently shared among individual program address spaces using dynamically linked shared libraries. Shared libraries work by deferring certain binding operations until the program is loaded and executed. This sharing provides efficient utilization of the computer systems memory, but deferring the binding time of class libraries introduces the risk of version incompatibility between library and client software. But deferring the binding time also provides the opportunity to improve software by introducing new functionality and fixing software defects.

This work builds upon the idea of evolvable classes introduced by [Ellis & Stroustrup] suggesting the use of tables of offsets to members. Andrew Palay further developed the ideas of evolvable classes in his ΔC++ system, [Palay]. Like ΔC++, we define certain compatible changes to a class library that will not require recompilation or changes to a client.

1.1 Compatible Evolution

The following example illustrates the relationship between the implementor of a C++ library, the developer of an application (or another library) who uses that library, and the end-user of the application.

9 References

[Bancilhon] F. Bancilhon, C Delobel, P. Kanellakis, *Building an Object Oriented Database System - The Story of O_2*. Morgan Kaufman, San Mateo CA, 1992.

[Bannerjee] J. Banerjee, W. Kim, H. Kim, H. Korth. “Semantics and Implementation of Schema Evolution in Object-Oriented Databases”, Proceedings of the ACM SIGMOD Conference, pages 311-322, 1987.

[Birrell & Nelson] A. D. Birrell & B. J. Nelson, “Implementing Remote Procedure Calls,” ACM Trans. Computer Systems, 2(1), February 1984

[Ellis & Stroustrup] M. Ellis & B. Stroustrup, *The Annotated C++ Reference Manual*, AW 1990.

[Goldstein] T. Goldstein, A. Sloane, M.Ball, A. Palay, “Supporting the Evolution of Class Definitions - Workshop Report” in OOPSLA’93 - Addendum to the Proceedings, to appear in SIGPLAN Notices, 1994.

[Hamilton & Radia] G. Hamilton and S. Radia, “Using Interface Inheritance to Address Problems in System Software Evolution”, Proceedings of the ACM Workshop on Interface Definition Languages, 1994.

[Harrison & Ossher] W. Harrison and H. Ossher “Extension by addition: Building extensible software.” Research Report RC16127, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 1990

[Lenkov] D. Lenkov, M. Mehta, S. Unni, “Type Identification in C++”, Proceedings of USENIX C++ Conference, Washington D.C. April 1991.

[OMG] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, August 1991, OMG Document Number 91.8.1.

[Palay] A. Palay, “C++ in a Changing Environment”, in Proceedings of the Usenix C++ Technical Conference, Portland, September 1992, pages 195-206.

[Penney & Stein] D. Penney & J. Stein, “Class modification in the GemStone object-oriented DBMS”, in Proceedings of the ACM OOPSLA Conference, pages 111-117, 1987.

[Richardson and Carey] J.E. Richardson and M.J. Carey, “Persistence in the E Language: issues and implementation,” *Software—Practice and Experience* number 19, pages 1115-1150, 1989

[SOM] *OS/2 2.0 Technical Library System Object Model Guide and Reference*, IBM Doc 10G6309.

[Stroustrup] B. Stroustrup, “Namespaces”, ANSI X3J16/93-0105, ISO WG21/N0312.

10 Acknowledgments

The authors gratefully acknowledge the help provided by Michael L. Powell of SunSoft and Michael S. Ball of SunPro, as well as all of the reviewers on the program committee. We would especially like to thank Brian T. Lewis for the thorough careful reading and editing of this paper.

Disadvantages of this approach are that it must be performed for *all* calls, and that it is difficult to recover after a failed check (possible actions are to call `abort` or raise an exception.)

(ii) "Migrate" the old `otbls` to new `otbls`.

In this case the entries for unsupported virtual functions are set to point to runtime error routines. This ultimately has the same error semantics as in option (i) , but there is no overhead of bounds checking for virtual functions. (In this way it is analogous to the use of thunks in a conventional `vtbl` to avoid unnecessary adjustments to the `this` pointer.)

(iii) Separate the notions of *compile-time type* and *run-time type*, and use the run-time `typeid` operator to give the programmer explicit control.

We observe that we wish the compile-time type system to treat all conformant versions of a type as the same (e.g. allow any version of the type `Bar` to be passed to a function declared as taking a parameter of type `Bar`), but that we may wish to distinguish between them at run-time. In other words applying the `typeid` operator to objects of different versions of a type would produce different results. The programmer could then explicitly test for version conformance and make appropriate decisions.

(iv) Reify the notion of namespace and allow it to dynamically correspond to a physical entity such as a dynamically linked shared library. For example create a construction which allows a namespace to be returned from a system level operation. `For example: namespace foo = dlopen(..);` Use the normal C++ type system to enforce conformance

At the present time we do not consider any of the approaches listed above to be satisfactory and we remain interested in exploring further approaches. Although (iv) is the most intriguing, we do not believe the C++ language can tolerate any additional major extensions.

The semantics we really require are those of the "newer is better" variety. A link time mechanism, such as that in SunOS 4.x, which supports a (major, minor) version pair to denote compatible versions provides the correct semantics. A change in major version denotes an incompatible change. A change in minor version denotes a compatible change, and all compatible changes are serialized i.e. there is no branching of compatible changes. The binary file resulting from the compilation and link-editing of a library or application records the version number of the dynamically linked library on which it depends. At run-time the dynamic linker selects according to its search rules a version of the library which is the "newest", which matches exactly in major version number, and for which the minor version number is greater than or equal to the minor version number specified in the client's dependency list. Thus there is only one version of a type present in the client's address space and it supports at least the functionality required by the client.

8 Conclusion

We have presented a new object model called the *Shared Object Model* and a new C++ implementation model called the *Object Binary Interface* (OBI. These models restrict some semantics of C++ programs to allow objects to span versions and to allow multiple implementations of an interface to co-exist within an address space. The shared object model supports virtual functions, public and protected data as well as derivation. The principal restriction over the normal C++ object model is that private data may only be accessed either explicitly or implicitly through the "this pointer." By using the linkage specification of C++ and allowing this key restriction of C++ usage, we are able to add an evolutionary version object model without the need for a language extension. Initial performance estimates indicate that this strategy is viable and adds little overhead to programs.

This is the OBI up level for both virtual and non-virtual inheritance: 4 instructions

```
ld    [%i0], %l1         ! get optr
ld    [%l1+A_IN_B_OFFSET],%l2 ! get the base class offset
add   %i0,%l2,%l4        ! add the base class offset to this
ld    [%l4],%l5          ! return the value
```

Table 5. Summary of Instruction Counts

	call up level non-virtual inheritance	call up level virtual inheritance	get an up level value non-virtual inheritance	get an up level value virtual inheritance
cfront	4	4	1	2
OBI	5	5	4	4

We believe that the overhead indicated is well within our acceptable levels. On modern multi-stage pipelined architecture, we believe that the actual execution time will be insignificant. Further study of the dynamic properties of the implementations are underway.

7 Object Version Migration

So long as the interface changes which resulted in the new version of a class are upward compatible, client code which was compiled against the old version works correctly when passed a "new" object. Although we recommend against exposing data members in software interfaces, it is still possible to have public and protected data members in this scheme. Public and protected data members are still present and are located at fixed offsets within the defining class. Our upward compatibility rules require that no public or protected members are reordered or deleted. Virtual functions retain the same index within the otbl which defines them (since upward compatible changes do not reorder or delete virtual functions).

A problem however arises when code compiled against the new version of the interface gets hold of an "old" object. How can this happen? This can arise if for example we have the following sequence of versions of an interface.

```
// version V1.0
class Bar { ... };
void Bar::f( Bar *pb ) { ... }

// version V1.1
class Bar { ... };
void Bar::f( Bar *pb ) { ... pb->g(); ... }
void Bar::g() { ... }          // new in version 1.1
```

If libraries implementing both versions of class `Bar`, V1.0 and V1.1, are linked into the program's address space, then "new" code (in V1.1) can be passed an "old" object (created by V1.0) and attempt to invoke an operation - such as `g()` - on it which the V1.0 object does not support.

At the present time we are exploring possible solutions to the version migration problem. Note that in the case where shared linkage is used only to support evolution, and only one version of a library is linked to a program at one time, there are no such "version migration" problems. Among the approaches we have considered are:

(i) Perform a runtime bounds check on the virtual function invocation, testing the `otbl` index against the value of `n_functions`. A similar check is necessary for access to public and protected data members.

```
struct __smfptr {
      size_t base_index;       // index of base
                               // -1 if direct
      size_t index;            // index of virtual
                               // negative if non-virtual
      fptr  faddr;             // only for non-virtual
};
```

Note that the offset of the `optr` need not be stored, since it is always at offset zero. The base index is used to calculate the adjustment to be applied to the `this` pointer, by a lookup in the `otbl`.

6 Instruction Counts

Most operations are identical to cfront. This includes getting data and calling virtual functions at the level of the type the data and function is defined at. The only differences occur when accessing data or calling a function in a base class. For the cfront implementation model, the are different code sequences for virtual and non-virtual inheritance. For the OBI implementation model, the code sequences are the same for virtual and non-virtual inheritance. Table 5 summarizes the instruction count difference.

6.1 Calling a virtual function defined in a base class.

This is case for current cfront non-virtual inheritance: 4 instructions

```
ld    [%i0+8],%l1              ! load the vptr into l1
ldd   [%l1+8],%l2,%l3          ! load double the thisDelta
                               ! and function addr into l2,l3
add   %l2,%o0,%o0              ! adjust the thisDelta
call  %l3
```

This is the case for current cfront virtual inheritance from a virtual base: 4 instructions

```
ld    [%i0+OFFSET_PB],%o0      ! load pB->PB into o0
ld    [%o0+8],%l1              ! load the vptr into l1
ld    [%i1+12],%l2             ! load the function addr into l2
                               ! the adjusted this pointer is
                               ! already in o0)
call  %l2
```

This is the case for OBI both virtual and non-virtual inheritance call up level: 5 instructions

```
ld    [%i0],%l1                ! load the optr into l1
ld    [%l1+A_IN_B_OFFSET],%l3 ! load B's A base into l3
ldd   [%l3+8],%l4,%l5          ! load double the thisDelta and
                               ! function addr into l4,l5
add   %i0,%l5,%o0              ! add the thisDelta to this
call  %l4
```

6.2 Accessing a member in a base class

Current cfront non virtual inheritance: 1 instructions

```
ld    [%i0+OFFSET_OF_MEMBER],%l1
```

This is the current cfront virtual inheritance: 2 instructions

```
ld    [%i0+OFFSET_PB],%l0            ! load B's A base into %l0
ld    [%l0+OFFSET_OF_MEMBER],%l1     ! indirect through PA
```

FIGURE 9. Virtual function call in shared linkage.

```
class A {
public:
      virtual void vfA1();
      virtual void vfA2();
};

class B : public A {
public:
      void vfA1();      // override
      virtual void vfB();
};

B* pB = new B;

pB->vfB();              // (1)
pB->vfA1();             // (2)
pB->vfA2();             // (3)
```

This is implemented approximately as follows. Note that we show the implicit `this` parameter for purposes of illustration only.

```
__otbl* ot = pB->optr;

function_part* fp - &(ot->function[1]);               // (1)
fp->*fct(this + fp->this_adj);

fp = &(ot->function[0]);                              // (2)
fp->*fct(this + fp->this_adj);

ot = ot->base[0].base;                                // (3)
fp = &(ot->function[1]);
fp->*fct(this + fp->this_adj);
```

Note that in the third case the client code references the *nested otbl* for class A in class B (nested tables are built even in the case of single inheritance). Thus in later versions derived classes can add overrides of base class virtual functions.

5.13 Overriding the return type of virtual functions

To implement calls of virtuals in which the return type has been overridden we use the same mechanism as for ordinary virtual function calls with the additional step of adding the return adjustment stored in `ret_adj`. Note that in fact *both* adjustments must always be done, since at run-time there is no knowledge of whether or not the function return type has been overridden.

This implementation does not support cases where the overriding contraviant return type is a class with shared linkage or is virtually derived from the original type. A full implementation requires the execution of a complex code sequence that calculates the return adjustment based on the actual type of the object.

5.14 Pointer to Member-Function

The pointer-to-member-function machismo is similar to the mechanism described in Section 5.5 on page 10. Pointers to member functions are represented by a structure `__smfptr`, defined as follows

FIGURE 8. Algorithm for the initialization of otbls

```
__otbl* MY_TYPE::otbl_creator_function()
{
__otbl *my_otbl ← allocate(sizeof(fixed_part) +
            n_bases * sizeof_(base_part) +
            n_functions * sizeof(function_part));
my_otbl->n_functions ← MY_TYPE_NUMBER_OF_FUNCTIONS;
my_otbl->n_bases ← MY_TYPE_NUMBER_OF_BASES;
my_otbl->size ← sizeof(MY_DATA) + sizeof(_otbl*);
my_otbl->most_widen_offset = 0;

//Size and assign position of all base classes
foreach direct base otbl_b of MY_TYPE in traversal order do
   {
   my_otbl->base_b ← call TYPE_b::otbl_creator_function();
   // unify previous allocated virtual base class
   foreach base otbl_bb of otbl_b
      {
      is otbl_bb already allocated virtually?
         {
            delete base_bb;
            base_bb ← previously allocated otbl;
         }
      else // Not seen yet. Allocate its region.
         {
            base_bb->most_widen_offset = - my_otbl->size;
            my_otbl->size += base_bb->size ;
         }
      }
   }
// Assign the values for the virtual function part
foreach base otbl_bbb both direct and indirect
   {
   foreach function_f in base_bbb
      {
      if MY_TYPE overrides function_f
         {
         base_bbb->function_f->fct ← function;
         base_bbb->function_f->this_adj ← 0;
         }
      else
         base_bbb->function_f->this_adj ← base_bbb.widen_offset;
      if function_f has a covariant return type == MY_TYPE
            base_bbb->function_f->ret_adj ← COVARIANT_TYPE_OFFSET_f ;
      }
   }
return my_otbl;
}
```

5.12 Virtual Function Calls

Calling a virtual function for a class with shared linkage involves an indirection through one or two `otbls`, depending on whether the function is defined in the derived class or inherited from a base class. Figure 9 below illustrates the implementation.

```
extern "shared" class A { public:int a1, a2;};
class B {
public: int b1;
      A ba;
      int b2;
} myB;
      myB.b2 = 999;
```

The offset of the member `b2` of `B` is dependent on the size of `class A`, which must be looked up in the `otbl`. So the code to make the assignment to `myB.b2` looks like

```
* (int*)((char*)&myB + OFFSETOF(B::ba) + myB.optr->size)) = 999;
```

5.8 Static Data Members

Static data members are unaffected by a "`shared`" linkage specification.

5.9 Static Class Functions and friend functions

For static class functions and friend functions, there is no access to the private data members. This is because these functions do not vector through the virtual function table. There is thus no way to ensure that the compile-time correspondence of the private data of the `this` pointer actually corresponds to the version of the implementation that is reached by these functions.

5.10 Constructors, operator new and sizeof

As in default linkage `operator new` is always called before the constructor. Note that allocating an object of a class with shared linkage always requires a call to `sizeof` no matter what the storage class the user specifies. Moreover in shared linkage `sizeof` is not a compile-time operator, but requires a lookup of the `otbl`. We can illustrate its implementation with pseudo-code as follows

```
size_t sizeof(TYPE) { return TYPE_otbl.size; }
```

But since there might be more than one implementation of type, it is more interesting to use `sizeof` on an instance.

```
size_t sizeof(void* inst)
{
      void* derived = ((char *)inst) + inst->_optr->most_widen_offset;
      __otbl* ot_of_derived = *((__otbl**) &derived);
      return ot_of_derived->size;
}
```

5.11 Initialization of Otbls

Since values in the `otbl` representing instance size, base object offsets, and virtual functions are not known until run-time (when the dynamic linker is run at program initialization time, or after a `dlopen`) the `otbl` cannot be initialized statically. However it must be initialized before any instance of the class can be allocated. In particular, the `otbl` must be allocated before any call to `sizeof`. Thus the `otbl` can be initialized by `.init` code in the shared object, or by code run dynamically the first time `sizeof` is called for the type and the result cached for further use.

The initialization of the `otbl` is done by calling a built-in static member function for each class with shared linkage, `otbl_creator_function`. Figure 9 below describes the otbl creation algorithm.

Unlike normal vtbls, there is no difference in the code generated for casts to virtual and non-virtual bases[1].In the example,

```
__otbl* ot = pC->optr;
```

sets `ot` pointing to `C_otbl`. Then

```
ot->base[1].offset // has the value 16.
```

Adding `16` to `pC` sets `pB` pointing to the object representing the `B` part of `C`, and the `optr` for that object points to `BinC_otbl`.

The cast to `V*` is implemented as follows

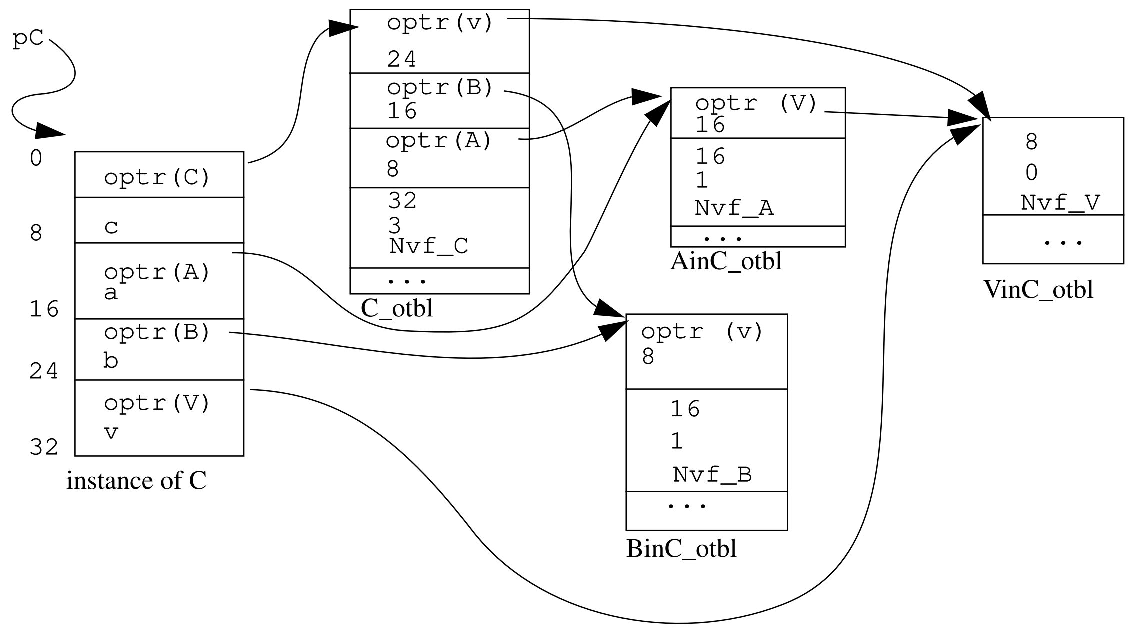
```
ot = pB->optr;
```

sets `ot` pointing to `BinC_otbl`.

```
ot->base[0].offset // has the value 8.
```

Finally, adding `8` to `pB` sets `pV` pointing to the object representing the `V` part of `B` (and of `A` and `C` since it's virtual), and the `optr` for that object points to `VinC_otbl`.

FIGURE 7. Layout of derived class instance and otbls



5.7 Composition

By *composition* we mean the declaration of a member of a class which is itself of a class type. This is the common use o embedding one class in another as in:

---

1. The GNU C++ compiler, `g++`, uses a similar scheme to implement virtual derivation in general where the virtual base pointers are replaced by offsets stored in the `vtbl`.

5.6 Derivation and Casts

Casts are implemented much as in conventional implementations, see for example [Ellis & Stroustrup] pages 221-227, except that the adjustments applied to pointers are not compile-time constants, but rather must be looked up in the object's `otbl`.

Downcasts (i.e. casts from base to derived class) are implemented by casting to the most derived class using the `most_widen` field, and then back to the intermediate class. As a consequence of this, downcasts from virtual bases are supported, but the result of downcasting to a non-virtual intermediate base which occurs twice in the hierarchy is undefined. This is the only difference we have identified between shared linkage and "standard" C++, and many users find it less restrictive than the absence of casts from virtual bases. In passing we note that this same restriction is also present in SOM's C++ mapping.

FIGURE 6. Cast from a derived class to a base class

```
extern "shared" {
      class V { public: int v; private: ... };
      class A : public virtual V {
            ...
      public:
            int a:
      private:
            ...
      };
      class B : public virtual V {
      public:
            int b;
      private:
            ...
      };
      class C : public A, public B {
      public:
            int c;
      private:
            ...
      };
}

C* pC = new C;
B* pB = pC;// implicit cast, B* ← C*
V* pV = pB; // implicit cast, V* ← B*
```

The layout of the object pointed to by `pC`, and it's network of `otbls` is shown in Figure 9 below. Pseudo-code illustrating the implementation of the casts is

```
__otbl* ot = pC->optr;
pB = (B*) ((char*)pC + ot->base[1].offset);
__otbl* ot2 = pB->optr;// B in C otbl
pV = (V*) ((char*)pB + ot2->base[0].offset)
```

protected members, since they are laid out in declaration order immediately after the `optr` and before the private members. We require that implementation code for the class, i.e. member function and friend function bodies, which directly accesses the private members must always be compiled against the up-to-date version of the class definition, and so the compile-time determined offsets are valid. To access a member defined in a base class[1] we proceed in the following steps

(i) follow the object's `optr` to its `otbl`

(ii) index in the `base` array to find the appropriate `base_part` structure

(iii) add the `widen_offset` to the address of the object to find the base class object

(iv) add the offset of the member within the base class.

Thus the C++ statement "`myB.a2= 999;`" is equivalent to the following C statement

```
((A*)((char*)&myB + myB.optr->base[0].offset))->a2= 999;
```

5.5 Pointers to Data Members

A pointer to data member is represented by an instance of class `__smdptr`, which is defined as follows:

```
struct __smdptr {
      size_t base_index;
      size_t offset;
};
```

The `base_index` is negative if the it a direct member, and not a member of a base class. An example of accessing a pointer-to-data-member is:

```
extern "shared" class B {
public:
      int b1:
      int b2;
};

extern "shared" class D : public B {
public:
      int d:
};

int D::* p = &D::b2;
D* pD = new D;
pD->*p = 999;
```

The pointer `p` is represented as

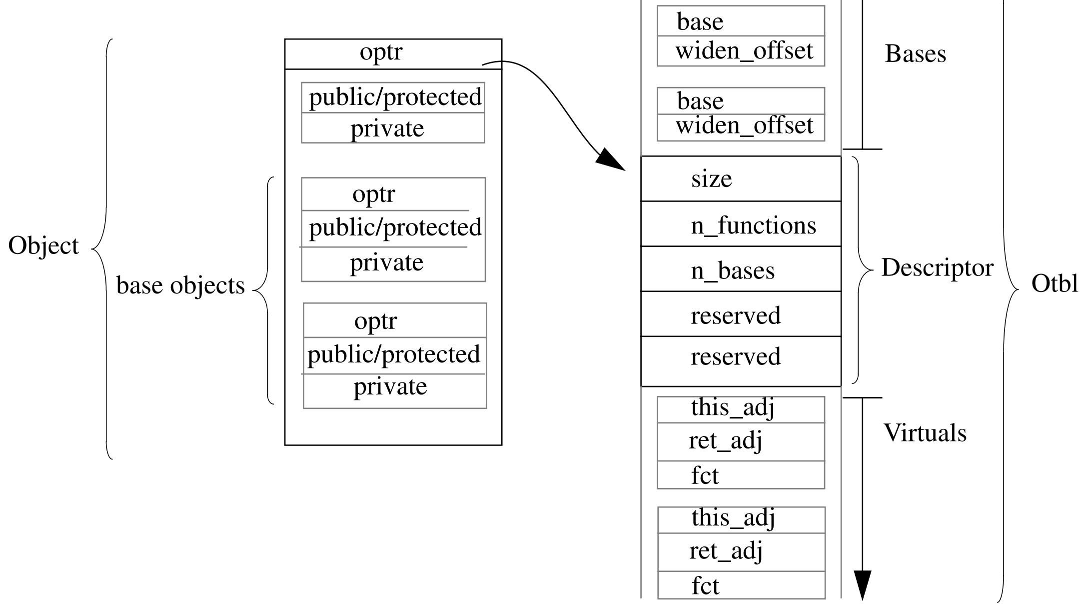
```
__smdptr p = {1,4};
```

Referencing through `p` results in a lookup of the appropriate `base_part` structure in the `otbl`

```
__otbl* ot = pD->optr;
char* pTmp =(B*)((char*)pD + (p.base_index < 0 ? 0 :
                  ot->base[p.base_index].widen_offset));
* ( (int*) ( pTmp + (char*)p.offset ) ) = 999;
```

1. In client code this must be a public or protected member of a public or protected base

FIGURE 4. Object and Otbl



5.3 Allocation of automatic and static instances

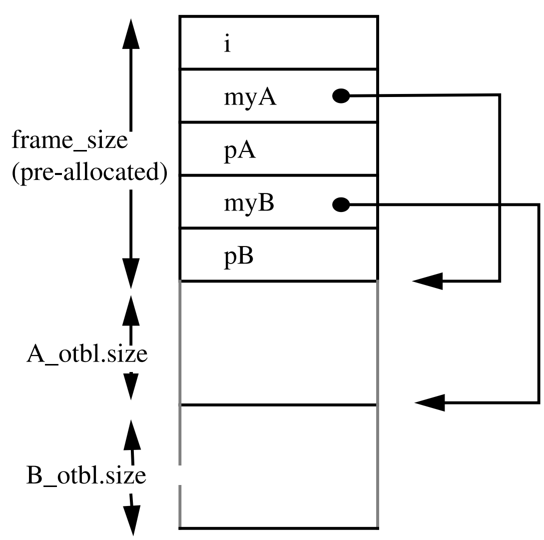
Automatic objects of a class with shared linkage are allocated on the stack frame via an `alloca`-like mechanism detailed below. Objects are represented by a pointer to a location elsewhere in the stack frame and the "real" object is laid out by special prologue code in the function. Static objects, whether global or file static, are represented by a pointer and the "real" object is on the heap since the real object is not known. Allocation and initialization of statics are done (as is conventional) with `.init` code.

FIGURE 5. Allocation of auto objects in Shared Linkage

```
extern "shared" {
      class A {
      public:
            int a1;
            int a2;
      };

      class B : public A {
      public:
            int b;
      };
}

void func() {
      int i;
      A myA, *pA;
      B myB, *pB;
}
```



5.4 Access to Data Members

Just as in default linkage, to access a member defined in the class itself (not a base class) we use an offset determined at compile-time. Notice that changing the size of the private part or extending the list of public and protected members does not change the offsets compiled into client code for existing public and

As with many simple ideas, these two rules generate many important implementation details. The rest of this paper describes the OBI implementation model, including several key algorithms.

5.1 Otbls

For classes with shared linkage, `otbls` take the place of `vtbls`. The terms `optr` and `otbl` originate from the `vptr` and `vtbl` described in [Ellis & Stroustrup]. The 'o' may stand for 'offset' or 'object'. Within an object, the public and protected data members immediately follow the `optr`. The object's `optr` points to the `size` field in the fixed part of the `otbl`. The `base_part` structures are indexed backwards from there, and `function_part` structures are indexed forwards. The `base_part` structures are indexed in order on a left-to-right, pre-order depth-first traversal of the inheritance DAG, with `base_part`'s for virtual bases left to the end and allocated in the order the virtual bases were encountered in the graph traversal. The `function_part` structures are allocated for each virtual function defined in the class in order of declaration.

An `otbl` is global data, and will have an external "mangled" name. It can be treated as a global variable or (probably better) as a private static data member of its class. One difference between `vtbls` and `otbls`, is that `vtbls` along the left linear tree walk of the derivation are concatenated. This concatenation is not possible with `otbls`, as it would prohibit growth in adding new virtual functions to the end.

5.2 Class Layout

Within the public/protected section and the private section the members are laid out in declaration order. The order in which the base classes are laid out is unimportant, since all accesses to inherited members are calculated through offsets found in the `otbl`. The `otbl` consists of three parts:

(i) an array of `base_part` structures which grows backwards providing base class information. Each `base_part` consists of a pointer to an `otbl` and the offset of the base class object

```
struct __otbl{
     struct base_part {
        __otbl* base;
        size_t widen_offset;
     } base[n_bases];
```

(ii) a fixed descriptor section describing the size of instances, the number of direct virtual functions, and the total number of base classes. This part also contains access to the runtime type information.

```
     size_t size;
     size_t n_functions;
     size_t n_bases;
     TypeInfo* tinfo;
     size_t most_widen_offset;
```

(iii) an array of `function_part` structures which consist of a pointer to a direct virtual function of the class and adjustments for the object pointer and the return value.

```
struct function_part {
        size_t this_adj;
        size_t ret_adj;
        void (*fct)();
     } function[n_functions];
```

The relationship between an object and its otbl is presented in Figure 4 on the next page. The otbl points to the "otbl descriptor" which is located in the middle of the object. In the following sections we show through pseudo-code and diagrams how classes with shared linkage and objects of such classes are used. This description covers *all* possible uses of the object by client code.

```
extern "C++" {
      void f(int);
      class A { ... };
      class B : public A { ... };
};
class C { ... };
extern "shared" {
      int g(X*);
      class X { ... };
      class Y : public X { ... };
};
```

The function `void f(int)` and the classes `A`, `B` and `C` have default or "`C++`" linkage; and the function g(X*) and the classes `X`, `and` `Y` have "`shared`" linkage. A given class can have only one linkage specification throughout a program (This is the same rule which applies to the linkage specification of functions in the C++ draft standard). Moreover the representation of a *pointer-to-member-of-shared-class* is different to the representation of a *pointer-to-member-of-default-class* and assignment of one to the other is not allowed. In addition there is a semantic restriction on how classes with different linkage specifications can be combined. A derived class and all its base classes must have the same linkage specification.

4.2 Semantics of the Shared Object Model

The principle new rule of the OBI is that the private data of an object must be strictly encapsulated. Thus the 1990's corollary of "only friends may touch your private parts is that "*no one else may touch your private parts*." The official rules of the OBI are:

(i) The only access to the private data members of an object is through the object's (explicit or implicit) `this` pointer. Non-virtual functions, both member functions and non-member friend functions, cannot access the private parts. The shared object model does not prohibit access to public or protected data members.

(ii) New public and protected members (both data-members and virtual functions) must be added to the end of the list of existing members.

(iii) The one definition rule for object types is relaxed for shared linkage types within a program (but not within a single compilation unit).

5 The OBI implementation model

The shared object model is implemented using the Object Binary Interface (OBI) implementation model. Of course, users do not need to know the mechanics of the OBI implementation model. The principle notion is that the OBI is a binary interface similar in spirit to the System V Application Binary Interface (ABI). It is possible that a future generation of the OBI may allow for binary compatible linkage of other Object-oriented languages besides C++. The OBI has three basic concepts:

(i) The OBI specifies that each instance of a class with shared linkage contains an `optr` (pronounced oh-pointer) as its first element. The `optr` points to an `otbl` which describes the layout of the class instance and how to invoke the virtual functions of the class.

(ii) Public and protected data members come first. After all public and protected data members, come the private data members. Consequently the public and protected data members of a class can be extended because only the private data is adjusted. But the total ordering of the public and protected data members is preserved!

(iii) All inheritance is implemented in a fashion similar to virtual inheritance. The fundamental mechanism is to allow the relationship between base and derived objects to change and evolve. The virtual inheritance mechanisms in `cfront` provide the inspiration for the necessary indirection.

4 The Shared Object Model

The Shared Object Model provides a C++ programmer with an additional annotation to partition class library into classes that the implementor guarantees never to change, and classes that may change. The first category often corresponds to “key” components of the library whose interface is extremely well understood and for which optimal implementations are well known. The overall run-time performance of the library is dominated by the performance of these key components. For example, an implementor of a reference counting object might decide to exposure via inline functions the implementation. The implementor is trading off the risk of the need to change the implementation of the reference counting object in order to gain better performance. Inlines effectively exports the implementation (e.g. the layout of the private members) of a class and so precludes any possibility of evolution.

The second category corresponds to outer layers of abstraction, usually having more complex semantics and larger granularity. Thus adding overhead to function calls for these layers will have less impact on the performance of the library as a whole. Correspondingly these are the layers whose semantics are most subject to change from version to version and where there is greater variability in implementation. They are the classes where support for evolution is most important. Support for evolution is not appropriate for all classes, but where library design makes a natural partitioning of classes such that evolution is both possible and effective for critical classes.

Recognizing this, we chose to add extra syntax to identify evolvable classes to the compiler. We considered using a `#pragma`, but instead chose to use a linkage specification: `extern "shared" { ... }`. A linkage specification fits very well with our requirements:

(i) It is not a language extension. We abhor any more extensions to this language. But the C++ draft standard allows an implementation to provide an arbitrary set of linkage specifications in addition to the required specifications `"C"` and `"C++"`. We have defined a specification `"shared"`, which results in linkage conventions suitable for long-lived interfaces and for linkage across shared object boundaries.

(ii) Code using linkage specifications is portable. Compilers which do not support a particular linkage specification will generate a warning, but supply default linkage.

(iii) It is semantically accurate. The concept of “linkage” is concerned with communication across translation unit and library boundaries, especially if it provides cross-language models.

(iv) It is syntactically appropriate. Entire header files can be easily “wrapped” to use shared linkage, just as existing C header files could be given C linkage with `extern "C"`.

4.1 extern “shared” { ... }

We chose the word *shared* because we wanted to allude to System V shared libraries (dynamically linked libraries), and because most of the suggested alternatives, such as `dynamic`, `global` and `export`, had other conflicting connotations. In our implementation a “`shared`” linkage specification affects only class definitions and their member functions and does not affect non-member functions. The layout of class objects and the mechanism for calling virtual functions depends on the *linkage specification* of the class. For example in the following code fragment:

2.3 Schema Evolution

The object-database community has long recognized the necessity for supporting type and schema evolution. Object-database systems that support evolution include: Orion [Bannerjee], GemStone [Penney & Stein] and O2 [Bancilhon]. The E compiler in the Exodus system [Richardson and Carey] follows a technique similar to the OBI in using offsets to point to virtual base classes. But no support for multiple versions of an object class in a database has been described in this work.

2.4 Design focus of the OBI

All the mechanisms that have been proposed to support evolution of class definitions either involve significant extra indirection at run-time or require extra relocation work by the runtime linker. Linker start-up latency is already noticeable in dynimcally-linked C++ programs. Our concern is that additional start-up latency would be unacceptable.

Unlike ΔC++ and SOM we believe that not all changes of a class interface are of equal utility between version release. During software development, radical rearrangement of function order and other class hierarchy is useful. There is little value in supporting arbitrary reordering of member functions between two *released* versions of a class. Our selections of changes and usage rules provide compatible evolution of types and interface-implementation independence with limited impact on C++ usage. The OBI design highest priority is to allow the private parts of an object to change arbitrarily. The second highest priority is to allow extension by addition [Harrison & Ossher]. Other changes such as rearrangement of class hierarchies are not unimportant, but are not the focus of this work. If desired, it is simple to add a #pragma annotation which specifies the order of data members. The OBI design differs from previous work in that it focuses on a select few high values changes and it supports multiple simultaneous implementations of a class within an address space.

3 Existing Support of versioning in C++

There are several techniques that might support versioning within C++ including renaming, derivation, and namespaces. All of these techniques provide some handle on the problem, but all are insufficient to meet our criteria stated above. Renaming functions, for example has been around as long there have been symbolic identifiers. Version 6 UNIX had a file seeking operating system call named `seek` which took one 16 bit integer argument. The obvious flaw of files exceeding 64 K was fixed in version 7 UNIX by adding in the operating system call named `lseek`. This works amazingly well because both the old and new operations can coexist. In practice there is no limit to the number of “l”’s or other version-specific characters one can add onto an identifier. But exceedingly long identifiers become unwieldy and are a blight on the code.

Objects require additional support. Users would like new objects to be accepted anywhere an old object was accepted. One solution, proposed in [Hamilton & Radia], is to represent version conformance explicitly by modeling it using derivation.

```
class UNIX_File_v6 {
      unsigned short seek(unsigned short offset, int whence);
};
class UNIX_File_v7 : public UNIX_File_v6 {
      unsigned long seek(unsigned long offset, int whence);
};
```

As an alternative the recently defined namespace extension [Stroustrup] to the ANSI/ISO C++ draft standard provides a mechanism that can alleviate the unwieldiness of the long identifiers. Applications that contain objects derived from the UNIX_File_v6 class must be edited and recompiled to take advantage of the newer UNIX_File_v7 class. We want a solution that allows application objects to transparently use the latest version of the class. The approach we have chosen makes versions orthogonal to the C++ type system.

version differences between XLib R15 and XLib R16 created objects. The code mechanisms of unmarshalling (and dynamic loading) arrange to get the correct addresses for the new library into the virtual function tables.

A single CADMaker application will contain instances derived from both R15 and R16 XLib code. Enabling such multiple versions can be easily achieved by strictly respecting the encapsulation boundary of objects. An object should not access the private data of any other object, even those of its own class, except through virtual function calls. The private data of an object is truly kept private. Thus the only restriction over the normal C++ object model is that private data may only be accessed either explicitly or implicitly through the "this pointer."

These usage rules place restrictions on what the library vendor can do with C++ objects. Despite these restrictions, the advantages of allowing multiple definitions of an object in the same address space at the same time is significant and worthwhile for many applications. Note also that even in ordinary non-distributed applications dynamic loading makes it possible to have multiple implementations of an object coexist in the same address space.We will refer to the property of supporting multiple implementations of a type within the one address space as *interface-implementation independence*.

2 Related work

Several previous authors have discussed the issue of versions and evolution, some in the context of programming languages, but most in relation to schema evolution in object-oriented databases. A brief summary of these approaches is described in [Goldstein].

2.1 ΔC++

ΔC++'s solution does not support multiple implementations of an object in an address simultaneously. ΔC++ allows a developer to make compatible changes to a class with minimal recompilation of clients. The set of compatible changes maintains or extends the class interface without altering the code sequence used to access members of that class. ΔC++ accomplishes this by resolving classes at link time. Changes are handled by extending the linker to support new relocation types such as member offset. Most code sequences are identical to `cfront` [Ellis & Stroustrup] generated code except for constructors/destructors, calls to non-virtual members, access to embedded structures and some optimizations. The expense of this technique is additional program start-up time latency while the linker processes the additional relocations.

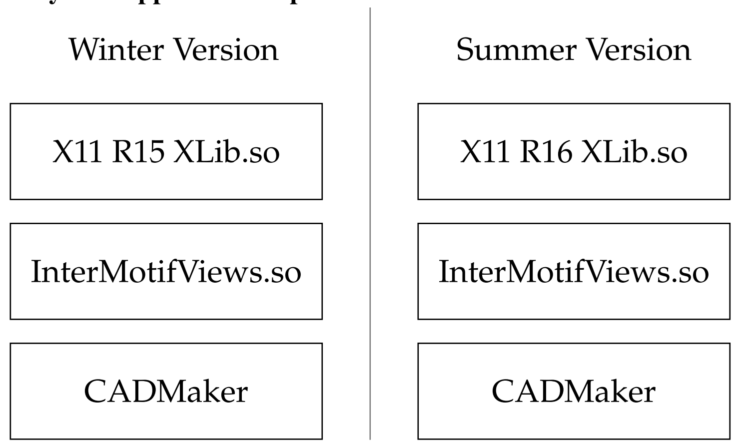
2.2 System Object Model

The System Object Model (SOM) was developed at IBM. There are several variants - SOM, DSOM that supports distributed objects within the general framework of the Object Management Group's CORBA, and PSOM that supports persistent objects. All of these provide run-time support for a common interface to "objects" independent of programming language. They also have an extensive metaclass framework, which in the DSOM case supports the dynamic invocation interface of CORBA.

By extending IDL's notion of interfaces, [OMG], with an implementation construct called "release-order:" SOM supports extension and change of an interface. Other aspects of class evolution are implemented by SOM's metaclass protocol. SOM puts the burden on the user to maintain the correct release order across versions. For example, the release-order entries must only be extended, never removed or deleted. SOM uses a dispatch mechanism that supports multiple inheritance but favors single inheritance implementations. SOM is a hybrid of the Smalltalk metaclass object protocol and the OMG object model layered on top of C++. Additional overhead is required for multiple inheritance involving caching and hash lookup tables. The principal difference between the OBI and SOM is that the OBI approach still assumes the essential mechanism and efficiency found in C++, while SOM emulates the object model using Smalltalk-like framework.

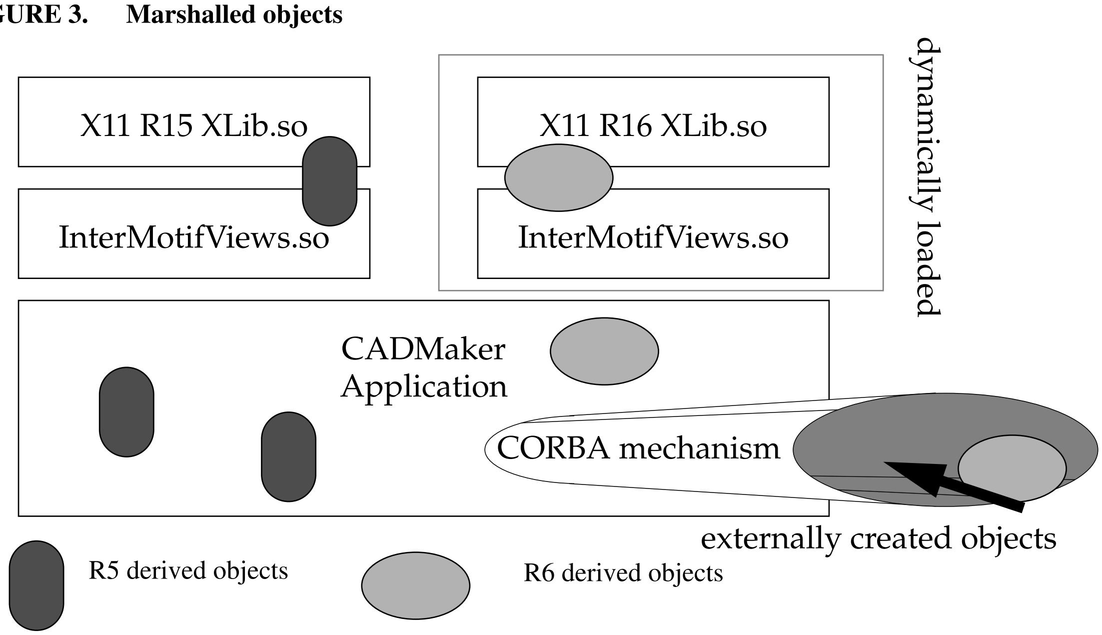
applications using InterMotifViews is a sophisticated network oriented application called *CADMaker*. Since XLib and InterMotifViews come from separate vendors, their will always be a time delay between release of the XLib library and the next release of InterMotifViews. It may even be possible that end-users receive updates of the X11 library before the maker of InterMotifViews sees it. Figure 2 depicts a Winter and following Summer rconfiguration of the CADMaker application.

FIGURE 2. Library and Application Dependencies



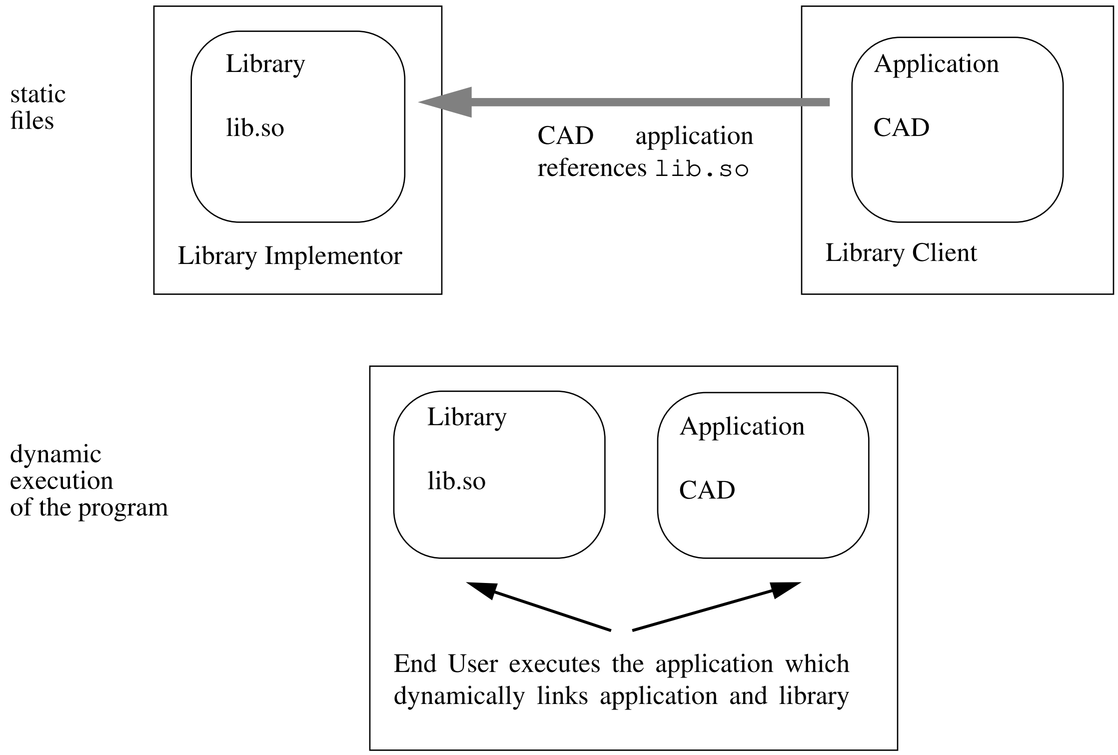
Suppose the environment and libraries supports multiple user distributed applications. The hypothetical InterMotifViews is designed for running workgroup applications which allow users to work together. As well, some servers at a CADMaker client site are running the Winter version and some are running the Summer version of XLib. CADmaker programs share objects. Objects which are made by compatible but different class libraries in an external program are transferred and shared from one address space to another through mechanisms such as CORBA. Figure 3 depicts the summertime execution context of CADMaker. where X11 R16 implementation objects come from an external address space marshalled through some RPC mechanism.

FIGURE 3. Marshalled objects



The correct behavior is that the application has access to both R15 and R16 libraries simultaneously. If XLib objects are accessed only through virtual functions, the CADMaker application should be unaware of the

FIGURE 1. Library, Application and End-User



Suppose the library implementor changes the implementation of `lib.so` or changes the interface in an upwardly compatible way to provide additional functionality and ships the new version to the end-user. The end-user expects - quite reasonably - that the expensive CAD package bought from the ISV will still run. The end-user cannot recompile the CAD package - she doesn't have sources, and may not even have a compiler. On the other hand the library-implementor, even if it's the platform library implementor, can't require all ISV's to synchronize with them. In reality, synchronization will be even more complicated since the CAD package likely requires libraries from several different vendors. To fulfill the promise of modular software components, object-oriented technology must support compatible replacement of software libraries.

It must be possible to introduce compatible changes to a library without requiring any changes to the source code or even recompilation of the clients of the library. Run-time mechanisms for object-oriented languages must support *compatible evolution* of class libraries.

1.2 Interface-Implementation Independence

Recent technologies such as the Object Management Group's CORBA [OMG] which support distributed applications allow objects to span address spaces. Objects may be defined in one program and used in another. Objects move from one address space to another through a linearization and communication process such as *remote procedure call* [Birrell & Nelson] or Object Invocation [OMG]. A very useful mechanism is for the external address space to pass an instance of a derived class. Frequently, during the unmarshalling of the object, it may be discovered that the new address space does not have the derived class implementation of the object, but only have the base class interface of the object. Dynamic linking provides a mechanism a program may use to acquire the corresponding implementation for a marshalled object.

Imagine, for example, that X11's *XLib* library is written in C++. Suppose there is a hypothetical third party library vendor writing a window system toolkit called *InterMotifViews* that uses XLib. Among the

The Object Binary Interface — C++ Objects for Evolvable Shared Class Libraries

Theodore C. Goldstein
Sun Microsystems Laboratories
ted.goldstein@eng.sun.com

Alan D. Sloane
SunPro
alan.sloane@eng.sun.com

Abstract

Object-oriented design and object-oriented languages support the development of independent software components such as class libraries. When using such components, versioning becomes a key issue. While various ad-hoc techniques and coding idioms have been used to provide versioning, all of these techniques have deficiencies - ambiguity, the necessity of recompilation or re-coding, or the loss of binary compatibility of programs. Components from different software vendors are versioned at different times. Maintaining compatibility between versions must be consciously engineered. New technologies such as distributed objects further complicate libraries by requiring multiple implementations of a type simultaneously in a program.

This paper describes a new C++ object model called the *Shared Object Model* for C++ users and a new implementation model called the *Object Binary Interface* for C++ implementors.These techniques provide a mechanism for allowing multiple implementations of an object in a program. Early analysis of this approach has shown it to have performance broadly comparable to conventional implementations.

1 Introduction

Software either evolves or dies. Software evolution occurs in response to numerous requirements including bug fixes, user demands for greater functionality, and especially to support changes in related software. Object-oriented programming promises to allow individual class library components to evolve independently of its clients. Many modern operating systems allow software libraries to be efficiently shared among individual program address spaces using dynamically linked shared libraries. Shared libraries work by deferring certain binding operations until the program is loaded and executed. This sharing provides efficient utilization of the computer systems memory, but deferring the binding time of class libraries introduces the risk of version incompatibility between library and client software. But deferring the binding time also provides the opportunity to improve software by introducing new functionality and fixing software defects.

This work builds upon the idea of evolvable classes introduced by [Ellis & Stroustrup] suggesting the use of tables of offsets to members. Andrew Palay further developed the ideas of evolvable classes in his ΔC++ system, [Palay]. Like ΔC++, we define certain compatible changes to a class library that will not require recompilation or changes to a client.

1.1 Compatible Evolution

The following example illustrates the relationship between the implementor of a C++ library, the developer of an application (or another library) who uses that library, and the end-user of the application.