Layered, Server-based Support for Object-Oriented Application Development

Guruduth Banavar   Douglas Orr   Gary Lindstrom
Department of Computer Science
University of Utah, Salt Lake City, UT 84112 USA
{banavar,dbo,lindstrom}@cs.utah.edu

Abstract

This paper advocates the idea that the physical modularity (file structure) of application components supported by conventional OS environments can be elevated to the level of logical modularity, which in turn can directly support application development in an object-oriented manner. We demonstrate this idea through a system-wide server that manages the manipulation of such components effectively. The server is designed to be a fundamental operating system service responsible for binding and mapping component instances into client address spaces.

We show how this model solves some longstanding problems with the management of application components in existing application development environments. We demonstrate that this model's effectiveness derives from its support for the cornerstones of OO programming: classes and their instances, encapsulation, and several forms of inheritance.

1 Introduction

In a traditional application development environment such as UNIX, application components ultimately take the form of *files* of various kinds — source, object, executable, and library files. Entire applications are typically built by putting together these components using inflexible, and sometimes ad-hoc, techniques such as preprocessor directives and external linkage, all managed via makefile directives.

It is also natural for application developers to generate components corresponding to incremental changes to already existing application components, especially if they subscribe to the software engineering principle known as "extension by addition." This principle holds that it is better to extend software not by direct modification, but by disciplined addition of incremental units of software. Advantages of "extension by addition" include better tracking of changes and more reliable semantic conformance by software increments. Most importantly, the increments themselves have the potential to be reused in similar settings.

This perspective leads one to conclude that traditional OS environments inadequately support component manipulation and binding for modern application development. Object-oriented (OO) programming offers a potential solution to this inadequacy. In OO programming, inheritance is a mechanism that aids in the effective management of software units and incremental changes to them. Indeed, in advanced OO languages, increments as well as base components have independent standing (e.g., "mixins"). Other aspects of OO programming, notably encapsulation, have demonstrated benefits to large-scale software development via enhanced abstraction. There is much to gain from supporting these features within the infrastructure of an application development environment, beyond whatever support is provided by the languages in which application components are written.

In this paper, we demonstrate a principled, yet flexible, way in which to effectively construct applications from components. This facility is orthogonal to makefiles, and does not impose new techniques for building individual application components. Instead, it relies on the idea that the *physical* modularity of traditional application components (i.e. files) can be endowed the power and flexibility of *logical* modularity. Such logical modules can then be manipulated using the concepts of *compositional modularity*, where first-class modules (defined in Section 3.1) are viewed as building blocks that can be transformed and composed in various ways to construct entire application programs. Individual modules, or entire applications, can then be instantiated into the address spaces of particular client processes. Compositional modularity has a firm foundation [4], and has been shown to be flexible enough to support several important effects and styles of object-oriented programming [2].

This approach has other advantages besides making system building more principled and flexible. First, it enables a form of OO programming with components written in non OO languages such as C and Fortran. Second, it enables *adaptive* composition, whereby the system that manages the logical layer can perform various composition-time, "exec"-time, and possibly run-time optimizing transformations to components. For example, system services (such as libraries) can be abstracted over their actual implementations, adding a level of indirection between a service and its actual implementation. This permits optimizations of the service implementation based on clients' disclosed behavioral characteristics. Such system-level support is explored elsewhere [14, 17, 15]; this paper focuses on application level support.

It is important to make clear that compositional modularity supported by a logical layer is not in conflict with object-orientation supported by component-level languages. For example, C++ programmers deal with two distinct notions of modularity: classes, fundamental to logical modularity, and source files, which deal with physical modularity. These two modularity dimensions share many characteristics, but have very different senses of composability, i.e. inheritance for classes, and linkage for files. Indeed, they are rather orthogonal in the minds of C++ programmers, because class definitions and source files do not always bear 1-1 relationships, and linkage is performed in a "class-less" universal namespace flattened by name mangling. In essence, they manage programs at two levels: classes with their semantic relationships, and files with their linkage relationships. With our approach, we accord physical artifacts (i.e. files) a degree of manageability comparable to that enjoyed by logical artifacts (i.e. classes).

In the following section, we present an application scenario that motivates the architecture presented in this paper. In Section 3, we present the layered architecture of our system, as well as the steps in constructing applications. Section 4 describes the functionality of the heart of the system. Section 5 presents specific solutions to the problems in Section 2. We then compare our work with related research, present our current status and envisioned future work, and conclude.

2 A Motivating Scenario

Consider a scenario in which a team of developers is building an image processing application using a vendor supplied (shrink-wrapped) library. Say the team completes building an initial version of the application (which is large-scale, say, greater than 100K lines of code), which is now ready for system testing. We can imagine several problems deriving from this scenario:

(i) Call wrapping. Suppose that the team finds that the application malfunctions because it calls a library function edge_detect() on an image data structure, consistently with an incorrect storage format, say with pixels represented as type FLOAT when BYTE was expected. Using traditional tools, this problem is rectified by inserting another library function call to the routine floattobyte() before each site in the application where edge_detect() was being called. This approach not only requires extensive modification of the application source code, but also expensive recompilation. Moreover, if two separate shrink-wrapped libraries are to be put together in this manner, sources might not even be available. Instead, it is more desirable to "wrap," at binding time, calls to edge_detect() with an adaptor that calls floattobyte(), all without recompiling the large application. However, such a facility is not usually supported in conventional OS environments.

(ii) Library extension management. Suppose further that the team decides that the application could work much better with an image format slightly different from the format expected by the library, but one which is easy to convert to and from the old format. If the new format is to be supported for future projects, it is best to change all library functions to accept and return the new format. However, sources for the library are not available, hence it cannot be directly modified. Thus, this would require developing and integrating a separate extension to the library. Furthermore, there could be several other independent extensions to the library that need to be integrated and supported for future applications. Developing such incremental extensions is much like subclassing in OO programming, but there is usually no support for effectively managing such incremental software units.

(iii) Static constructors and destructors. Imagine that the team wants to make sure that all statically defined images are properly allocated and initialized from disk before the program starts, and flushed back to disk before the program terminates. Currently available techniques for doing this are difficult and cumbersome.

(iv) Flat namespace. Say the image processing library uses the Motif library, which is in turn implemented in terms of the lower-level X library. Thus, in the traditional scenario, all the symbols imported from the Motif and X libraries become part of the interface exported by the image library. There is no way to prevent clients of the image library from obtaining access to the lower level library interface, or possibly suffer name collisions with that interface.

The system architecture we present in the following sections offers an effective solution to the above problems. Specific solutions to these problems are given in Section 5.

3 Architecture

3.1 Conceptual Layering

The first step in presenting an architecture for managing object modules is to clarify the conceptual layering of application components.

Conceptually, artifacts of physical modularity, i.e. files of various kinds, form a *physical* layer. These modules may be written as components in conventional languages that have no notion of objects. For example, in the case of C, there is no support for manipulating physical modules, much less for generating and accessing instances of them at run time — files are simply a design-time structuring mechanism.

The physical layer is managed with the help of traditional programming language environments. For example, the C language preprocessor, compiler, the make utility, the debugger, and library construction utilities help the programmer to develop application components of various kinds.

In the architecture presented here, each physical module can be manipulated as a first-class compositional module in what we shall conceptualize as the *logical* layer. In this layer, construction of entire applications is directed by scripts written by application programmers describing the composition of logical modules. Scripts are written in a module manipulation language that supports not only a simple merge of modules in the manner of conventional linking, but also many others including attribute encapsulation, overriding, and renaming. Most importantly, since modules are first-class entities in this language, individual operations can be composed in an expression-oriented fashion to produce composite effects such as inheritance in OO programming.

The logical layer is managed by a special tool, in the design of which the following requirements were laid out. First, the tool must provide a language processing system for the module manipulation language. Second, it must perform essential operating system services: that of linking modules and loading them into client address spaces. Third, since these services are in the critical path of all applications, it must be able to perform optimizations such as caching. Finally, it must be continually available. For these reasons, the logical module layer in the prototype described here is managed by a server process — a second generation implementation of a server named OMOS [16].

The OMOS server is described in more detail in Section 3.3. The module manipulation language supported by OMOS is derived from the programming language Scheme[6], and is based on the module manipulation language Jigsaw[4]. The model supported by this language, called compositional modularity, and its impact on developing applications in an OO manner, are explored in Section 4.

3.2 Application Construction

In this section, we describe the steps in constructing an application, based on the architecture depicted in Figure 1.

The first step is to build individual application components (physical modules) using a conventional programming language environment. (In this discussion, we shall consider only C language components, although the same ideas can be applied to another language such as Fortran.) Individual components, such as c1.c, c2.c, and c3.c in Figure 1 can be designed as traditional program files with no knowledge of the logical layer. Alternatively a component can be designed to be reused via suitable programming in the logical layer, such as a "wrapper" module described in Section 4.

Application components may be owned and managed by the user or by OMOS. In Figure 1, c1.c, c2.c, and c3.c are user provided application components. System provided components, such as libraries, are owned and managed by the OMOS server and accessed via service requests to OMOS.

The second step is to create a *module spec*, a file that describes the creation and composition of logical modules from application components. This is written in a *module language* described in Section 4. In Figure 1, app.ms is a user module spec that describes how to put the components of the application together. Module specs can themselves be modular; they can refer to other module specs. For example, app.ms may refer to libc, a system provided module spec that describes how to put together the components of a standard system library with a client module.

The final step is to request the module server to execute the module spec and instantiate (i.e. load) the result into a client address space. Module specs may be executed by calling a stand-alone version of OMOS from within a makefile, and the loading step performed interactively.

3.3 The OMOS Server

As mentioned earlier, OMOS is a continuously running process (a server) that is designed to provide a linking and loading facility for client programs via the

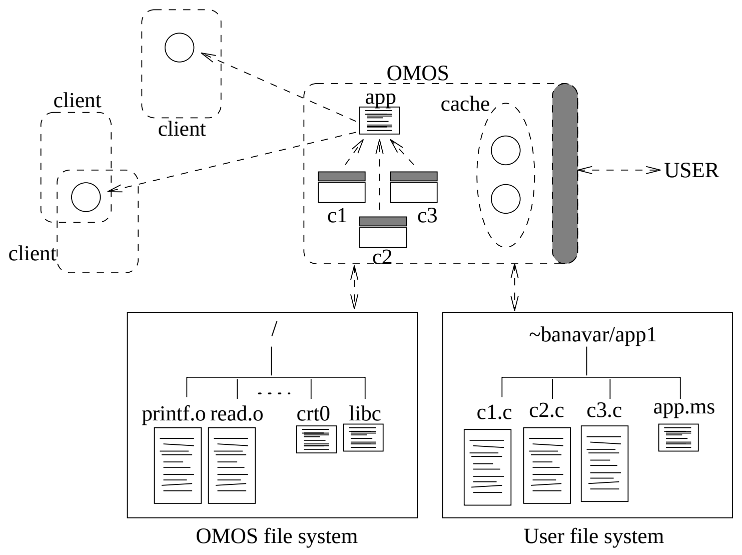


Figure 1: Overall architecture. c1.c, c2.c, etc. are user application components to be composed as described in the user module spec app.ms. Printf.o, etc., are system components to be composed according to module specs libc, etc. These components are composed by OMOS, possibly cached, and instantiated into client address spaces. The user can directly interact with OMOS via a command line interface to effect module composition and instantiation.

use of module combination and instantiation. OMOS supports three main functions: execution of module specs to compose applications, caching of intermediate results, and program loading. Module specs and their execution is described in the rest of the paper. The other two functions are described briefly below; the details, given in [16, 14], are beyond the scope of this paper. Additionally, use of meta-protocols with OMOS is explored in [15].

Evaluation of a module expression will often produce the same results each time. As a result, OMOS caches module results in order to avoid re-doing unnecessary work.

Since OMOS loads programs into client address spaces, it can be used as the basis for system program execution (the "exec server") and shared libraries [14], as well as dynamic loading of modules. Combining a caching linker with the system object loader gives OMOS the flexibility to change implementations as it deems necessary, e.g. to reflect an updated implementation of a shared module across all its clients.

This concludes a general description of the architecture of our system. In the following section, the functionality provided by the system as exported through the module language is described.

4 Module Management

As argued in Sections 1 and 2, an infrastructure that aims to support effective application development must support the flexible management of application components. We further argued that the management of components, their extensions, and their bindings is essentially similar to the management of classes and subclasses via inheritance in OO programming. This argument behooves us to demonstrate that our architecture does indeed support the essential concepts of OO programming, viz. classes and inheritance, which we show below in Sections 4.1 and 4.2 respectively.

Given the facilities described in this section, it is in fact possible to consider doing OO programming with a non OO language (such as C). However, it is not possible to do full-fledged OO programming in such a manner, since the base language does not support first-class objects (see Section 4.1.3). Neither is it desirable, since OO language support (such as C++) might be directly available. Thus, the facility we describe here is intended mainly for enhancing application component management rather than for actual application programming.

4.1 Classes

In the model of compositional modularity, a *module* corresponds to a distillation of the conventional notion of classes [4]. A module is a self-referential scope, consisting of a set of defined and declared attributes with no order significance. Definitions bind identifiers to values, and declarations simply associate identifiers with types (defining a label subsumes declaring it). Every module has an associated interface comprising the labels and types of all its visible attributes. An important characteristic of modules is the self-reference of attribute definitions to sibling attributes (see [7] for details). Modules can be adapted and composed using operators that manipulate the interface and the corresponding self-references. Furthermore, modules can be instantiated, at which time self-reference is fixed, and storage allocated for variables.

4.1.1 Modules

An object file (".o", or dot-o file), generated by compiling a C source file, corresponds directly to a module as described above. A dot-o consists of a set of attributes with no order significance. An attribute is either a file-level definition (a name with a data, storage or function binding), or a file-level declaration (a name with an associated type, e.g. `extern int i;`)[1]. Such a file can be treated just like a class if we consider its file-level functions as the methods of the class, its file-level data and storage definitions as member data of the class, its declarations as undefined (abstract) attributes, and its `static` (file internal linkage) data and functions as encapsulated attributes. Furthermore, a dot-o typically contains unresolved self-references to attributes, represented in the form of relocation entries.

Symbols, both defined and merely declared, of physical modules make up the interface of logical modules. (For simplicity of presentation, we consider interfaces to comprise only the symbol names, without their programming language types; see [3] for a study of typed interfaces.) Compiled code and data in the actual object file represents the module implementation.

A physical dot-o is brought into the purview of the logical layer by using the primitive `open-module` in our module language. The syntax of this primitive is given in Figure 2. Once it is thus transformed into a logical module, it can be manipulated in several ways using the other primitives given in the figure, which are described in the following sections.

4.1.2 Encapsulation

Module attributes can be encapsulated using the operator `hide` (see Figure 2). However, in the case of C language components, encapsulation partly comes for free, since C supports the internal linkage directive, `static`. However, attributes can be hidden after the fact, i.e. non-static C attributes can be made static retroactively, with `hide`. This is a very useful operation as demonstrated in Section 5.

[1]Type definitions (e.g. struct definitions, and typedef's in C) are not considered attributes.

(open-module ⟨*path-string-expr*⟩)
(fix ⟨*module-expr*⟩ ⟨*section-locn-list*⟩)
(hide ⟨*module-expr*⟩ ⟨*sym-name-list-expr*⟩)
(merge ⟨*module-expr1*⟩ ⟨*module-expr2*⟩ ...)
(override ⟨*module-expr1*⟩ ⟨*module-expr2*⟩ ...)
(copy-as ⟨*module-expr*⟩ ⟨*from-name-list-expr*⟩
  ⟨*to-name-list-expr*⟩)
(rename ⟨*module-expr*⟩ ⟨*from-name-list-expr*⟩
  ⟨*to-name-list-expr*⟩)

Figure 2: Syntax of module primitives

Many OO systems support the notion of a *class* consisting of public and private (encapsulated) attributes. In our system, a similar concept of classes is supported by a Scheme macro `define-class`, with the following sytax:

(define-class ⟨*name*⟩
  ⟨*dot-o-file*⟩ ⟨*superclass-exprs*⟩ ⟨*encap-attrs*⟩)

For example, given a dot-o `vehicle.o` that contains, among other attributes, a global integer named `fuel` and a global method `display` that displays the value of the `fuel` attribute, one can write the following expression (within a module spec) to create a class named `vehicle` by encapsulating the attribute named `fuel`:

```
(define-class vehicle "vehicle.o" () ("fuel"))
```

This macro expands into the following simple module expression:

```
(define vehicle
   (hide (open-module "vehicle.o") '("fuel")))
```

4.1.3 Instances

As mentioned earlier, instantiating a module amounts to fixing self-references within the module and allocating storage for variables. In the case of instantiation of dot-o modules, fixing self-references involves fixing relocations in the dot-o, and storage allocation amounts to binding addresses. These two steps are usually performed simultaneously. Thus, a dot-o can be instantiated into an executable that is bound ("fixed") to particular addresses and is ready to be mapped into the address space of a process. Dot-o's can actually be instantiated multiple times, bound to different addresses. Consequently, fixed executables are modeled as instances (objects) of dot-o modules (classes).

A module is instantiated using the primitive `fix` shown in Figure 2. The argument ⟨*section-locn-list*⟩ specifies constraints for fixing the module to desired sections of the client address space.

A concept closely associated with first-class objects in conventional OO languages is *message sending*. (For example, in the classical Smalltalk sense, objects communicate by sending messages to each other.) However, as mentioned earlier, there is no notion of first-class objects at the physical layer, which is where physical modules are implemented using component-level languages. Thus, message sending is not directly supportable in our framework. However, we envision extending our approach to support a form of message sending via inter-process communication, as described in Section 7.

4.2 Inheritance

We now arrive at the central aspects of our model. In this section, we introduce the inheritance related primitives supported by the module language, and describe the manner in which they can be composed. We start by introducing the following four primitives whose syntax is given in Figure 2.

The primitive merge combines modules which do not have conflicting defined attributes, i.e. attributes with the same name. This semantics is analogous to traditional linking of object files. However, the idea here is to go beyond traditional linking and support other operations basic to inheritance in OO programming, such as the following.

The primitive override produces a new module by combining its arguments. If there are conflicting attributes, it chooses ⟨*module-expr2*⟩'s binding over ⟨*module-expr1*⟩'s in the resulting module.

The primitive copy-as copies the definitions of attributes in ⟨*from-name-list-expr*⟩ to attributes with corresponding names in ⟨*to-name-list-expr*⟩. The *from* argument attributes must be defined.

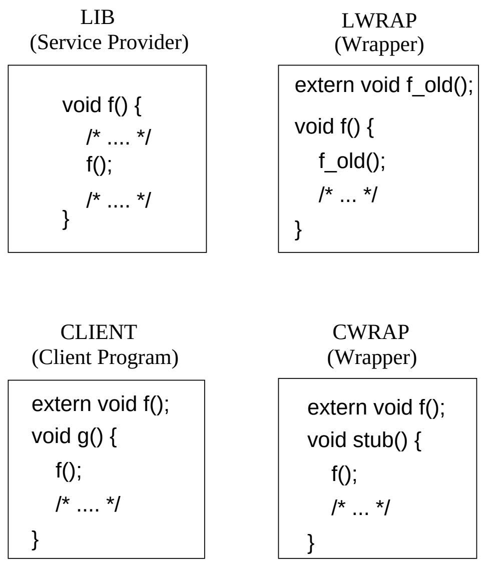
The primitive rename changes the names of the definitions of, *and* self-references to, attributes in ⟨*from-name-list-expr*⟩ to the corresponding ones in ⟨*to-name-list-expr*⟩.

To illustrate the use of the above primitives, the following section describes how to achieve several variations of a facility generally referred to as "wrapping."

4.2.1 Wrapping

Figure 3 shows a service providing module LIB with a function f(), and its client module CLIENT that calls f(). Three varieties of wrapping can be illustrated with the modules shown in the figure.

(1) A version of LIB that is wrapped with the module LWRAP so that all accesses to f() are indirected through LWRAP's f() can be produced with the expression:



```
void f() {
    /* .... */
    f();
    /* .... */
}
```

```
extern void f_old();
void f() {
    f_old();
    /* ... */
}
```

```
extern void f();
void g() {
    f();
    /* .... */
}
```

```
extern void f();
void stub() {
    f();
    /* ... */
}
```

Figure 3: Examples of wrapping.

```
(hide (override (copy-as LIB f f_old) LWRAP) f_old)
```

By using copy-as instead of rename, this expression ensures that self-references to f() within LIB continue to refer to (the overridden) f() in the resultant, and are not renamed to f_old.

(2) Alternatively, a wrapped version of LIB in which the definition of and self-references to f() are renamed can be produced using the expression:

```
(hide (merge (rename LIB f f_old) LWRAP) f_old)
```

This might be useful, for example, if we want to wrap LIB with a wrapper which counts only the number of external calls to LIB's f(), but does not count internal calls.

(3) If we want to wrap only a particular client module without wrapping the service provider, we can use the following expression:

```
(hide (merge (rename CLIENT f stub) CWRAP) stub)
```

In this case, renaming the client module's calls to f() produces the desired effect, since the declaration of f() as well as all self-references to it must be renamed.

Generalizing the above cases, the three varieties of wrapping possible in our model are shown pictorially in Figure 4. The leftmost column of the figure shows the given modules M1 and M2 and their wrappers W1 and W2. The top row shows a technique referred to as *method wrapping*, and the bottom row *call wrapping*. Box (a) corresponds to example (1) above, box (b) to (2), and box (d) to (3) above.

A technique known as before-after methods is used in the CLOS language to interpose calls to code be-

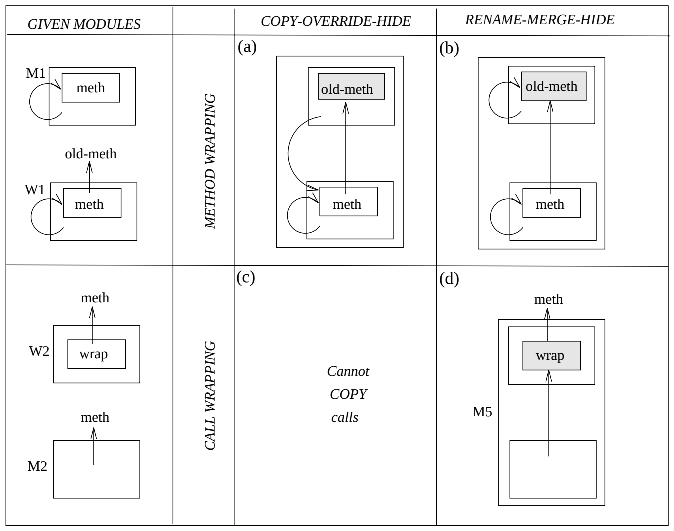


Figure 4: Wrapping scenarios. The leftmost column shows the given modules: M1 to be wrapped by W1, and M2 to be wrapped by W2. The top row shows the operations and effects of performing method wrapping, and the bottom row shows call wrapping.

fore or after a particular method proper. The above notions of method wrapping and call wrapping can be extended to support calling of precompiled routines by generating and wrapping the appropriate adaptors. For example, to call a method bef in module B before a method meth in module M, we can generate a wrapper module W with a function meth that first calls bef, and then calls the old definition of meth as old-meth. The modules M, W, and B can be combined in a manner similar to method wrapping to get the effect of a before-method:

```
(hide (override (copy-as M meth old-meth)
                (merge W B))
      old-meth)
```

4.2.2 Single and Multiple Inheritance

The idioms shown in Figure 4 are in fact the basis of inheritance in current day OO languages [2]. In this section, we give a brief idea of how these idioms can be used to achieve notions of inheritance.

Recall from Section 4.1.2 that a class can be defined using the macro define-class, which expands to a module expression that uses open-module and hide. A vehicle class was defined there. Using the same macro, a class can also inherit from another existing class.

For example, suppose a dot-o land_chars.o is created, which contains a global constant integer called wheels, and a function called display() that first calls a declared method called super-display(), then prints the value of wheels. Given such a module, a land-vehicle class can be created as a subclass of the previously defined vehicle module by writing:

```
(define-class land-vehicle "land_chars.o" (vehicle) ())
```

This macro expands to the module expression:

```
(define land-vehicle
  (hide (override (copy-as vehicle '("display")
                                    '("super_display"))
                  (open-module "land_chars.o"))
        '("super-display")))
```

In this expression, a module with attributes wheels and display is created, and is used to override the superclass vehicle in which the display attribute is copied as super_display. The new display method can access the shadowed method as super_display. In general, all such conflicting attributes are determined by a meta-level primitive called conflicts-between, and copied to a name with a super_ prefix. The copied super_display attribute is then hidden away to get a module with exactly one display method in the public interface, as desired. An important point here is that calls to display within the old vehicle module and the new land-vehicle module are both rebound to call the display method of the land-vehicle module.

The above idea of single inheritance can be generalized to multiple inheritance as found in languages such as CLOS [13]. In these languages, the graph of superclasses of a class is linearized into a single inheritance hierarchy by a language provided mechanism. A similar effect can be achieved with the define-class macro, except that the programmer must explicitly specify the order of the superclasses, as shown below:[2]

```
(define-class land-chars "land_chars.o" () ())
(define-class sea-chars "sea_chars.o" () ())
(define-class amphibian
    "amphibian.o" (land-chars sea-chars vehicle) ())
```

With the module operations supported by our module language, several other single and multiple inheritance styles can be expressed as well — these are described in [2].

5 Solving Old Problems

Using the operations defined on modules it is possible to conveniently solve long-standing problems in software engineering, encountered when using C, or C++. Several of these problems had solutions previously, but they were ad-hoc and/or required changes to source code. Module operations permit general solutions that impose no source code changes.

In this section, we delineate clean solutions to each of the problems enumerated in Section 2, in the same order.

(i) Wrapping calls. To solve the first problem of Section 2, the module spec for the image processing application can be written as given in Section 4.2.1, under call wrapping. Calls to edge_detect() can be wrapped with a wrapper method that first calls the function floattobyte() and then calls the edge_detect() library function.

(ii) Library extension management. The image library can be thought of as an OO class, and incremental changes to it can be thought of as subclasses that modify the behavior of their superclasses. The subclasses can be integrated with the superclass by means of a module spec that uses the notions of inheritance illustrated in Section 4.2.

(iii) Static constructors and destructors. In C++, there is a need to generate calls to a set of static constructors and destructors before a program starts. Special code is added to the C++ front end to generate calls to the appropriate constructor and destructor routines. However, the order in which such static objects are constructed is poorly controlled in C++ and leads to vexing environment creation problems for large systems.

Under some variants of Unix, the C language has handled the need for destructors in an ad-hoc fashion, by allowing programs to dynamically specify the names of destructor routines by passing them to the atexit() routine. In other variants, the destructors for the standard I/O library are hard-coded into the standard exit routine. In neither case is there any provision for calling initialization routines (e.g., constructors) before program startup.

In both the cases of C and C++, module operations allow addressing the problem by using a general facility, rather than special-purpose mechanisms. As shown in Section 4.2.1 as before-after methods, module expressions can easily be programmed to generate a wrapper main() routine that calls all of the initialization routines in the desired order, then call the real main() routine. Similarly, the exit() routine can be wrapped with an exit routine that calls all the destructors found in the module before calling the real exit().

(iv) Flat namespace. A longstanding naming problem with the C (and, to some extent C++) language has traditionally been the lack of depth in the program namespace. C has a two-level namespace, where names can be either private to a module, or known across all modules in an application.

With module operations, these problems can be avoided. Once a module that implements low-level functionality has been combined with a module that implements higher-level functionality, the functions in the former's interface can be subjected to the hide operation to avoid conflicts or accidental matches at higher levels.

6 Comparison to Related Research

This work is in essence a general and concrete realization of a vision due to Donn Seeley [20]. Although programmable linkers exist, they do not offer the generality and flexibility of our system.

A user-space loader such as OMOS is no longer unusual [19, 8]. Many operating systems, even those with monolithic kernels, now use an external process to do program loading involving shared libraries, and therefore linking. However, the loader/dynamic linker is typically instantiated anew for each program, making it too costly for it to support more general functionality such as in OMOS.

Utilities exist, such as dld [11], to aid programmers in the dynamic loading of code and data. These packages tend to have a procedural point of view, provide lower-level functionality than OMOS , and do not offer the control over module manipulation that OMOS provides. The dld utility does offer dynamic unlinking of a module, which OMOS currently does not support. However, since OMOS retains access to the symbol table and relocation information for loaded modules, unlinking support could be added.

The Apollo DSEE [1] system was a server-based

[2]Explicit specification of linearization is more useful than an implicit, language provided mechanism, see [2] for details.

system which managed sources and objects, taking advantage of caching to avoid recompilation. DSEE was primarily a CASE tool and did not take part in the execution phase of program development.

Several architecture description languages have been proposed, such as RAPIDE [12], the POLYLITH Module Interconnection Language (MIL) [5, 18], and OMG's Interface Definition Language (IDL) [9]. These languages all share the characteristic that they support the flexible specification of high-level components and interconnections. Our approach offers the important advantage that OO like program adaptation and reuse techniques (inheritance, in all its meanings) can be applied to legacy components written in non-OO languages.

An environment for flexible application development has been pursued in the line of research leading to the so-called subject-oriented programming (SOP) [10]. In this research, a "subject" is in essence an OO component, i.e. a component built around an OO class hierarchy. Subjects can be separately compiled, and composed using tools know as "compositors" (similar to OMOS). Compositors use various operators similar to the ones presented here. The primary difference between SOP and our research is that SOP is broadly conceived around the OO nature of individual components, and aims to build a toolset and object file formats specifically tailored for SOP. On the other hand, our research has focussed on layered evolutionary support.

7 Current Status and Future Work

OMOS is currently about 17,000 lines of C/C++ code. OMOS also uses the STk version of Scheme (11,000 lines) and the Gnu BFD object file library. OMOS runs on i386 and HP/PA-RISC platforms under the Mach operating system.

A foreseeable point of future work is to be able to support message sending, as described in Section 4.1.3. We have a design for converting static calls to IPCs. The basic idea is that a module instance corresponds to a thread in an address space. (Thus one can have many module instances within the same address space.) With this, message sending between instances is modeled as IPC, by converting static calls to IPC calls. For example, the expression

```
(msg-send m1 foo m2 bar)
```

wraps the static call to foo() within m1 with an IPC stub that calls the bar() routine within an instance of m2, which is itself wrapped with a receiving IPC stub. The crucial question here is that of determining the identity of the receiving instance of m2. One answer to this question is to have the msg-send routine also generate a constructor function that establishes the IPC environment between m1 and m2. For example, the constructor routine for m2 registers instances of m2 with a name service, and invocations of m1's foo() look up the identity of an m2 instance and establishes an IPC handle using that name. The particular instance of m2 that the name service returns can either be constant for the duration of the program, or be programmatically controlled from within base language modules.

Currently, OMOS is treated as a shared resource and module specs must be installed in an OMOS-accessible place by a trusted party. In order to provide the full functionality of the OMOS environment to users without opening security holes, we envision extending OMOS to work in a multi-level mode. OMOS will maintain system module scripts and cached executables. An individual user will run a limited version of OMOS that can generate cacheable instances from their own modules scripts or access global instances cached by the system. In general, this will distribute the computational load; the "system OMOS" will act primarily as a cache manager which seldom regenerates cached entities. Users engaged in application development will produce more caching activity, but it will be restricted to their own caches. User file and process activity will be limited to those resources to which the user naturally has access to. This will allow opening the full set of I/O primitives to user access without fear of abuse, and users will be unable to adversely affect one another via denial of service.

8 Conclusions

We have argued that application environments supported by conventional operating systems lack support for the effective management of application components. We illustrate that the problems faced by application builders are similar to those that are solved by the concepts of OO programming. We thus conclude that it is beneficial to support OO functionality within the component manipulation and binding environment.

We show that support for OO development can be achieved by elevating the physical modularity (i.e. separately compiled files) of application components to the level of logical modularity, managed by a system-wide server process. The server supports a module language based on Scheme, using which first-class modules can be manipulated via a powerful suite of operators. Expressions over modules are used to achieve various OO effects, such as encapsulation and inheritance, thus directly supporting application development in an OO manner. Furthermore, the server is designed to be a fundamental operating system service responsible for mapping module instances into client address spaces. In this manner, we enable a superior application development environment within a conventional operating system infrastructure.

Acknowledgments.

We gratefully acknowledge much implementation work on OMOS by Jeff Law. We thank Jay Lepreau and Nevenka Dimitrova for support and several helpful comments on this paper. This research was sponsored by the Defense Advanced Research Projects Agency (DOD), monitored by the Department of the Army under Grant number DABT63-94-C-0058, and by the Department of the Navy under Grant number N00014-91-J-4046.

References

[1] Apollo Computer, Inc, Chelmsford, MA. *DOMAIN Software Engineering Environment (DSEE) Call Reference*, 1987.

[2] Guruduth Banavar and Gary Lindstrom. Object-oriented programming in Scheme with first-class modules and operator-based inheritance. Technical Report UUCS-95-002, University of Utah, February 1995.

[3] Guruduth Banavar, Gary Lindstrom, and Douglas Orr. Type-safe composition of object modules. In *Computer Systems and Education*, pages 188–200. Tata McGraw Hill Publishing Company, Limited, New Delhi, India, June 22-25, 1994. ISBN 0-07-462044-4. Also available as Technical Report UUCS-94-001.

[4] Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *Proc. International Conference on Computer Languages*, pages 282–290, San Francisco, CA, April 20–23, 1992. IEEE Computer Society. Also available as Technical Report UUCS-91-017.

[5] John R. Callahan and James M. Purtilo. A packaging system for heterogeneous execution environments. *IEEE Transactions on Software Engineering*, 17(6):626–635, June 1991.

[6] William Clinger and Jonathan Rees. Revised^4 report on the algorithmic language scheme. *ACM Lisp Pointers*, 4(3), 1991.

[7] William Cook and Jen Palsberg. A denotational semantics of inheritance and its correctness. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 433–444, 1989.

[8] Robert A. Gingell. Shared libraries. *Unix Review*, 7(8):56–66, August 1989.

[9] Object Management Group. The common object request broker: Architecture and specification. Draft 10 Rev 1.1 Doc # 91.12.1, OMG, December 1991.

[10] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of OOPSLA Conference*, pages 411 – 428. ACM Press, September 1993.

[11] Wilson Ho and Ronald Olsson. An approach to genuine dynamic linking. *Software— Practice and Experience*, 21(4):375–390, April 1991.

[12] Dinesh Katiyar, David Luckham, and John Mitchell. A type system for prototyping languages. In *Proc. of the ACM Symp. on Principles of Programming Languages*, pages 138–150, Portland, OR, January 1994. ACM.

[13] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, Cambridge, MA, 1991.

[14] Douglas Orr, John Bonn, Jay Lepreau, and Robert Mecklenburg. Fast and flexible shared libraries. In *Proc. USENIX Summer Conference*, pages 237–251, Cincinnati, June 1993.

[15] Douglas B. Orr. Application of meta-protocols to improve OS services. In *HOTOS-V: Fifth Workshop on Hot Topics in Operating Systems*, May 1995.

[16] Douglas B. Orr and Robert W. Mecklenburg. OMOS — An object server for program execution. In *Proc. International Workshop on Object Oriented Operating Systems*, pages 200–209, Paris, September 1992. IEEE Computer Society. Also available as technical report UUCS-92-033.

[17] Douglas B. Orr, Robert W. Mecklenburg, Peter J. Hoogenboom, and Jay Lepreau. Dynamic program monitoring and transformation using the OMOS object server. In *The Interaction of Compilation Technology and Computer Architecture.* Kluwer Academic Publishers, February 1994.

[18] James M. Purtilo. The POLYLITH software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, January 1994.

[19] Marc Sabatella. Issues in shared libraries design. In *Proc. of the Summer 1990 USENIX Conference*, pages 11–24, Anaheim, CA, June 1990.

[20] Donn Seeley. Shared libraries as objects. In *Proc. USENIX Summer Conference*, Anaheim, CA, June 1990.