Layered, Server-based Support for OO Application Development

Guruduth Banavar
Douglas Orr
Gary Lindstrom

UUCS-95-007

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

April 14, 1995

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 process that manages a separate logical layer of components. The server is designed to be a central operating system service responsible for mapping component instances into client address spaces.

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

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

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 process that manages a separate logical layer of components. The server is designed to be a central operating system service responsible for mapping component instances into client address spaces.

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

Paper Category: Full Paper.

1 Introduction

An important tenet of software engineering holds that it is better to extend software not by direct modification, but by controlled 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 other similar settings.

In object-oriented (O-O) programming, inheritance is a mechanism that supports the effective management of incremental changes to software units. Indeed, in advanced O-O languages, increments as well as base components have independent standing (e.g., "mixins"). Other aspects of O-O programming, notably encapsulation, have demonstrated benefits to large-scale software development via enhanced abstraction. Hence there is much to gain from supporting these features within the infrastructure of an operating system, beyond whatever support is provided by the languages in which application components are written.

This perspective leads one to conclude that traditional OS environments support these concepts inadequately for modern application development. In such an environment, application components ultimately take the form of *files* of various kinds — source, object, executable, and library files. It is also natural for developers to generate components corresponding to incremental changes to

*Primary contact author. E-mail: *banavar@cs.utah.edu*, Phone: +1-801-581-8378, fax: +1-801-581-5843.

already existing application components, as a result of "extension by addition." However, 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. Moreover, it is becoming increasingly common that application programmers must deal with components in "shrink-wrapped" form, without sources to modify. Thus, we consider it imperative to advance currently available techniques for component manipulation and binding at the system level.

In this paper, we demonstrate a principled, yet flexible, way in which to construct applications from components. Our facility is orthogonal to makefiles, and we do not impose new techniques for building individual application components. Instead, we rely on the idea that the *physical* modularity of traditional application components (i.e. files) can be elevated to a separate layer of *logical* modularity. At this logical module layer, we apply 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 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 O-O programming with components written in non O-O languages such as C and Fortran. Second, it enables *adaptive* composition, where 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 [15, 18, 16]; we focus on application level support in this paper.

It is important to mention 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 support a similar degree of manageability for physical artifacts (i.e. files) as for logical artifacts (i.e. classes).

In the following section, we present a problem scenario that motivates the solution 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), and is now ready for system testing. We can imagine common problems deriving from this scenario:

1. 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 image type, say with pixels represented as type BYTE when FLOAT was expected. Using traditional tools, this problem is rectified by inserting another library function call to the routine **bytetofloat()** 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 **bytetofloat()**, all without recompiling the large application. However, such a facility is not usually supported in conventional OS environments.

2. 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 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 needs to be integrated and supported for future applications. Developing such incremental extensions is much like subclassing in O-O programming, but there is usually no support for effectively managing such incremental software units.

3. 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.

4. Say the IP 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 IP library. There is no way to prevent clients of the IP 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.

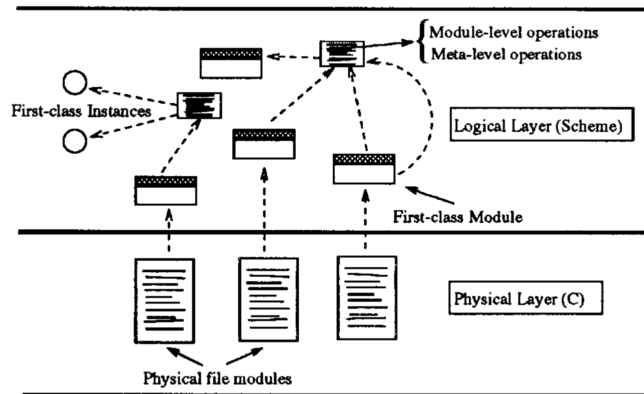

Figure 1: *Physical and Logical layers.* File modules in the physical layer are elevated to first-class modules in the logical layer (boxes with shaded interfaces). These modules are manipulated at the logical layer using scripts (boxes with text) to create new modules or instances. (Dashed arrows stand for transformations to modules.)

3 Architecture

3.1 An Object-Oriented Layer

We begin by showing how to elevate artifacts of physical modularity, i.e. separately compiled object files (".o" files), into first-class compositional entities. An entity is known as *first-class* if it can be stored in variables, as part of data structures, and passed into and out of functions.

The key is to introduce a separate layer on top of physical modules, as shown in Figure 1. The *physical layer* consists of physical modules. 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 — modules are simply a design-time structuring mechanism. Furthermore, module interconnection is specified statically via undefined attributes with external linkage, to be resolved in a pre-determined manner by a linker.

In order to make the above framework truly compositional, and flexible at the same time, we map each physical module to a first-class module in a *logical layer.* Attributes, both defined and merely declared, of physical modules make up the interface (shown as shaded portions of module boxes in Figure 1) of logical modules. For simplicity of presentation, we consider interfaces to comprise only the names of attributes, without their programming language types (see [3] for a study of typed interfaces). Compiled code and data in the actual object file defines the module implementation. Instances of modules are the actual mappings of program fragments into the address spaces of individual client processes.

In this system architecture, effective management of the logical layer becomes an important requirement. First, we need a language to express module manipulation, and an associated language processing system. Second, the system 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. The more the system knows about the behavior of the entire system, the more it can optimize, hence it is advantageous for it to have system-wide purview. Finally, it must be continually available. For these reasons, the logical module layer in our prototype is managed by a server process — a second

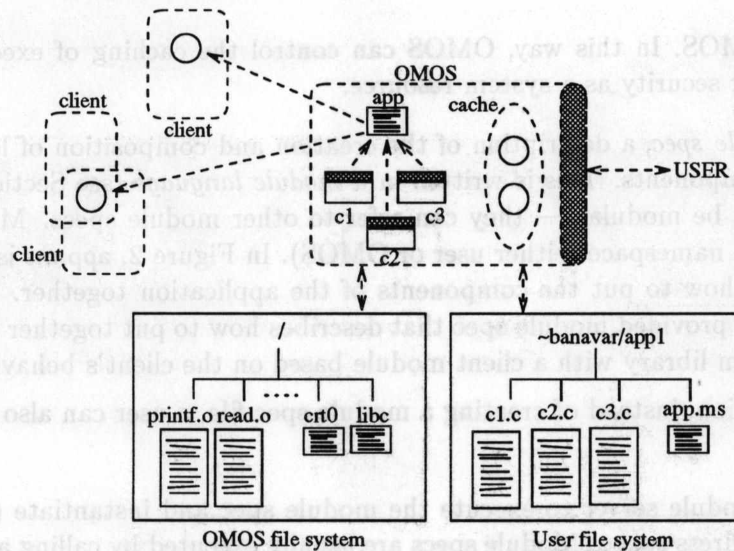

Figure 2: *Overall Architecture.* c1.c, c2.c, etc. are user application components to be composed as described in the user module spec app.ms. Printf.o, etc., are system components to be composed as given in module specs crt0, etc. These components are composed by OMOS, possibly cached, and instantiated into client address spaces. The user can directly interact with OMOS via a command line interface to effect module composition and instantiation.

generation implementation of a server named OMOS [17].

Our module language is derived from the programming language Scheme[6], and is based on the module manipulation language Jigsaw [4]. This language supports a simple merge of modules in the manner of conventional linking, as well as many others including attribute encapsulation, overriding, and renaming (see Section 4). But 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 O-O programming[2]. The impact of this idea on developing applications in an O-O manner is explored in Section 4.

3.2 Application Construction

In this section, we describe the steps in constructing an application, based on the architecture shown graphically in Figure 2.

1. Build individual application components using a conventional programming language[1]. Individual components can be designed either as traditional program files with no knowledge of the logical layer, or as components that can be "reused" via suitable programming in the logical layer. We will give examples of each case in a later section.

   Application components may be owned and managed by the user or the system. In Figure 2, 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

[1] In this paper, we consider only C language components. The same ideas can be applied to another language such as Fortran, but our system does not support it at this point. Also, we do not deal with inter-language components, although this is a point of interesting future work.

requests to OMOS. In this way, OMOS can control the caching of executable images, and guarantee their security as a system resource.

2. Create a *module spec*, a description of the creation and composition of logical modules from application components. This is written in a *module language* (see Section 4). Module specs can themselves be modular — they can refer to other module specs. Module specs are files in a file system namespace (either user or OMOS). In Figure 2, **app.ms** is a user module spec that describes how to put the components of the application together. On the other hand, **libc** is a system provided module spec that describes how to put together the components of a standard system library with a client module based on the client's behavioral characteristics.

   As another option, instead of creating a module spec file, a user can also specify composition interactively.[2]

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

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

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 O-O programming. This argument behooves us to demonstrate that our architecture does indeed support the essential concepts of O-O 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 do O-O programming with a non O-O language (such as C). However, it is not possible to do full-fledged O-O programming in such a manner, due to the reasons given in Section 4.1.3. Neither is it desirable, since O-O 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 framework 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

[2] In addition, old-style linking specs are supported for ease and backward compatibility. These are automatically translated to the module language.

(open-module ⟨*path-string-expr*⟩)
(fix ⟨*section-locn-list*⟩ ⟨*module-expr*⟩)
(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 3: Syntax of module primitives

(see [7] for details). Modules can be transformed and composed using operators that manipulate the interface and the corresponding self-reference. Furthermore, modules can be instantiated, at which time self-reference is fixed, and storage allocated for variables.

4.1.1 Modules

An object (".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;)[3]. 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.

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 3. Once it is thus viewed as a logical module, it can be subjected to several transformations and compositions using 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 3). 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 in Section 5.

Many O-O systems support the notion of a class consisting of public and encapsulated attributes. In our system, a similar concept of classes is supported by a Scheme macro **define-class** that expands into a module expression using **open-module** and **hide**. For example, given a dot-o **vehicle.o** that contains, among other attributes, a global integer named **fuel** and a global method **display**, one can write the following expression (in a module spec) to create a class named **vehicle** by encapsulating the attribute named **fuel**:

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

```
(define-class vehicle "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, an object file 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 be instantiated multiple times, bound to different addresses. Hence, fixed executables are modeled as instances of dot-o's. A module is instantiated using the primitive fix (see Figure 3).

A fixed executable is internally represented as an address map. An address map is a collection of entries that specify the address in the virtual memory of a process that a block in an object file is mapped to. To actually map a fixed executable, an application invokes the OMOS function **exec**, passing a specification of which fixed executable to map. OMOS uses the address map to guide where to remotely map regions into the client task. On systems that are not able to perform remote mapping operations, OMOS returns sufficient information that the client can do the mapping itself.

A concept closely associated with first-class objects in conventional O-O languages is *message sending*. 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

As mentioned earlier, an operation analogous to traditional linking can be accomplished via the primitive **merge** (see Figure 3). The primitive does not permit combining modules with conflicting defined attributes, i.e. attributes that are defined to have the same name. However, we go beyond traditional linking and support other operations basic to inheritance in O-O programming.

We will introduce three new primitives. The primitive **override** (see Figure 3) 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** (see Figure 3) 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*⟩.

4.2.1 Single and Multiple Inheritance

In our module language, a module can inherit from another by using the Scheme macro **define-class** introduced earlier. For example, given a dot-o **land_chars.o** which contains a global constant integer called **wheels**, and a function called **display**, a module called **land-vehicle** can be created as a subclass of the previously defined **vehicle** module by writing:

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

The **display** method within the dot-o **land_chars.o** overrides the original **display** method of the **vehicle** module. In addition, the new method can access the shadowed method as **super_display**. 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 macro expands into a module expression. 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**. 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.

The above idea of single inheritance can be generalized to multiple inheritance as found in languages such as CLOS [14]. 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:[4]

```
(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 — these are described in [2].

4.2.2 Wrapping

Variations on a facility generally referred to as "wrapping" are very useful for the purposes of flexible application building. Three varieties of wrapping are described below, and shown pictorially in Figure 4.

(i) *Method wrapping.* This is similar to the O-O notion of single inheritance as given in Section 4.2.1 above, and is explained in the top row of Figure 4. Two modules are given: **M1** defines a method **meth**, and has self-references to it; **W1** also defines a method **meth** in terms of a method **old-meth**, with possible self-references. We want to wrap **M1**'s **meth** with **W1**'s **meth**. This can be achieved in two ways as shown in boxes (a) and (b) of Figure 4. The module expressions are:

(a) (hide (override (copy-as M1 meth old-meth) W1) old-meth)
(b) (hide (merge (rename M1 meth old-meth) W1) old-meth)

The distiction between (a) and (b) has to do with the way self-references within **M1** are manipulated. The common case of wrapping uses the expression (a).

(ii) *Call wrapping.* Individual function *calls* can also be wrapped, as shown in Figure 4(d). Two modules are given: M2, which calls a method **meth**, and W2 which defines a method **wrap** that also calls **meth**. We want M2's call to **meth** to be indirected via W2's **wrap** method. This can be achieved via the module expression:

```
(hide (merge (rename m1 meth wrap) m2) wrap)
```

(iii) *Before-after methods.* The terms "before" and "after" methods are used in the CLOS language to refer to behavior that is called before or after a particular method proper. The above

[4] Explicit specification of linearization is superior to an implicit, language provided mechanism, see [2] for details.

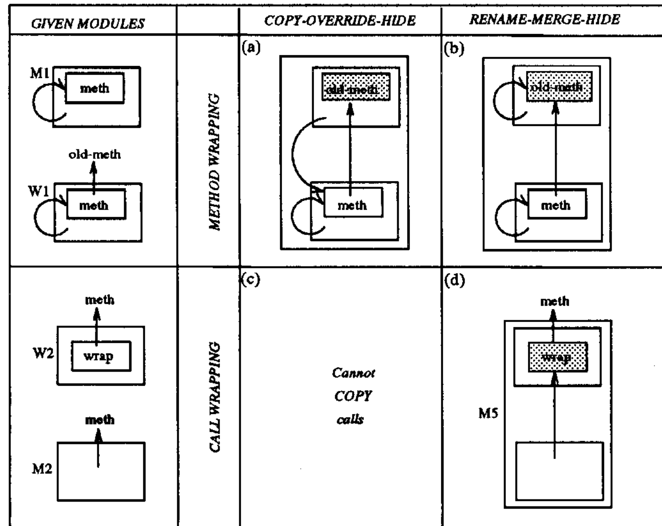

Figure 4: *Wrapping.* The leftmost column shows the given modules: **M1** to be wrapped by **W1**, and **M2** to be wrapped by W2. The top row shows the operations and effects of performing method wrapping, and the bottom row shows call wrapping.

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. The expression is shown below:

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

5 Solving Old Problems in Better Ways

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.

1. *Wrapping calls.* To solve the first problem of Section 2, the module spec for the image processing (IP) application can be written as given in Section 4.2.2, under call wrapping. Calls to **edge_detect()** can be wrapped with a wrapper method that first calls the function **bytetofloat()** and then calls the **edge_detect()** library function.

2. *Library extension management.* The IP library can be thought of as an O-O 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.

3. *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 of generating calls to initialization or termination routines by using a general facility, rather than special-purpose mechanisms. As shown in Section 4.2.2 as before-after methods, module expressions can easily be programmed to generate a wrapper main() routine that calls all of the initialization routines found within that module, 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(). In fact, the mechanism can be used to generate arbitrary constructor/destructor wrappers, permitting new entry/exit semantics to be defined for any routine in any module (not just the application main program).

4. *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. As a result, if an application uses library l1.a which imports symbols from another library l2.a, all symbols imported from l2.a are known by the application and become part of its exported interface.

   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 by Donn Seeley [22]. 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 [21, 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 [12], 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 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 definition languages (ADLs) have been proposed, e.g., RAPIDE [13], the POLYLITH Module Interconnection Language (MIL) [5, 20], 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 O-O like program adaptation and reuse techniques (inheritance, in all its meanings) can be applied to legacy components written in non-O-O languages.

An environment for flexible application development has been pursued in the line of research leading to the so-called subject-oriented programming [19, 10, 11]. However, the emphasis of this line of research has been new language design for application programming, rather than 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 forseeable 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 one thread in an address space, thus one can have many instances of a module in the same address space. With this, message sending between instances is modeled as IPC, by converting static calls to IPC calls. For example,

```
(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.

8 Conclusions

In this paper, 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 O-O programming. We thus conclude that it is beneficial to support O-O functionality within the component manipulation and binding environment.

We show that support for O-O development can be achieved by elevating the physical modularity (i.e. separately compiled files) of application components to a separate layer 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 O-O effects, such as encapsulation and inheritance, thus directly supporting application development in an O-O manner. Furthermore, the server is designed to be a central 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.

Acknowledgements.

We gratefully acknowledge much implementation work on OMOS by Jeff Law. We thank him as well as Jay Lepreau and Nevenka Dimitrova for support and several useful comments on this paper.

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. Attaching instance variables to method realizations instead of classes. In *Proc. International Conference on Computer Languages*, pages 291–299, San Francisco, CA, April 20–23, 1992. IEEE Computer Society.

[11] 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.

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

[13] 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.

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

[15] 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.

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

[17] 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.

[18] 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.

[19] Harold Ossher and William Harrison. Combination of inheritance hierarchies. In *OOPSLA Proceedings*, pages 25–40, October 1992.

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

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

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

Last modified on April 14, 1995.