UUCS-92-033 **July, 1992**

OMOS — An Object Server for Program Execution

Douglas B. Orr Robert W. Mecklenburg

Department of Computer Science, University of Utah
Salt Lake City, UT 84112
dbo@cs.utah.edu, mecklen@cs.utah.edu

Abstract

The benefits of object-oriented programming are well known, but popular operating systems provide very few object-oriented features to users, and few are implemented using object-oriented techniques themselves. In this paper we discuss a mechanism for applying object-oriented programming concepts to program binding (linking) and execution. We describe OMOS, an object/meta-object server that embodies a flexible object framework. The OMOS framework projects an object-oriented structure onto programs and shared libraries that may not have been originally developed for use within an object-oriented environment. This framework provides natural facilities for inheritance, interposition, and overloading of operations, as well as development of classes with dynamically evolving behavior.[1]

1 Introduction

In recent years object-oriented programming has gained widespread support due to its facilities for controlling modularity, division of responsibility, support for code reuse, and scalability[21]. We believe these features can be profitably applied to the problem of program binding and execution to achieve a more elegant solution than is currently available, while also providing increased functionality. We present a mechanism for applying object-oriented programming concepts to program binding (linking) and execution. We also describe the implementation of OMOS, an object/meta-object server which implements these concepts as a process in the Unix[2] operating system.

Current technology for constructing programs from modules is clumsy and lacks structure[24]. This clumsiness results in inefficiency which manifests as poor use of programmer skills, poor locality of program reference, poor reuse of existing code, poor use of virtual address space, and poor use of cpu time (i.e., avoidable recalculations). This lack of structure is, of course, not without reason. Historically, there has not been a structure which seemed sufficiently comprehensive and robust to encompass current techniques while providing the increased functionality of the object-oriented paradigm. Also, the large investment in existing techniques makes moving to an incompatible structure costly.

We believe that object-oriented programming concepts can be applied to existing techniques such that many of these inefficiencies can be overcome while still taking advantage of existing technology. We begin the discussion with a review of object concepts and how they relate to programs and name binding. We then discuss recent work which clarifies the relationship between modules and inheritance. After these preliminaries we will describe the features an object server must possess and a sample architecture. This discussion is followed by a more detailed discussion of a prototype implementation. Finally, we will review some results and make some observations on the concept of the OMOS system.

2 Objects and modules

An object is a collection consisting of some member data (called *slots*) and some member functions operating on that data (called *methods*). In this way an

[1] This research was sponsored by Hewlett-Packard's Research Grants Program and by the Defense Advanced Research Projects Agency (DOD), monitored by the Department of the Navy, Office of the Chief of Naval Research, under Grant number N00014–91–J–4046. The opinions and conclusions contained in this document are those of the authors and should not be interpreted as representing official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the U.S. Government, or Hewlett-Packard.

[2] UNIX is a trademark of AT&T.

Appears in identical form in *Proc. 2nd International Workshop on Object Orientation in Operating Systems*, Paris, France, September 1992.

object forms a self referential recursive scope. This scope can be seen by examining methods that use slots without actually naming the object being referred to, i.e., these references are implicitly qualified by the object (e.g., "self"). Methods implicitly exist within the scope of an object making such references unnecessary. Objects are instances of *classes* which resemble structure declarations extended to include functions. Class declarations describe the number and type of the slots and the number and type signatures of the methods. Classes can be combined through single or multiple inheritance.

Modern programming language theory distinguishes between *subtyping* and *inheritance*. A type t_2 is a subtype of t_1, if t_2's interface is compatible with t_1's. That is, it is legal to use an instance of t_2 wherever an instance of type t_1 is required. Inheritance, on the other hand, is an implementation technique which allows the combining of classes. Subtyping is often bundled with inheritance, blurring the distinction between the two concepts. In popular object-oriented programming languages (OOPLs) such as C++[11] public inheritance is the only mechanism available for implementing subtyping. An instance of a derived class can be used wherever an instance of a base class is required. This relationship is commonly referred to as an *isA* relationship.

The effect of (public) inheritance is to merge the data and method declarations of the participating classes into a single new class. Often, the classes being combined with inheritance contain duplicate member names. The treatment of member name collisions varies in popular OOPLs. In the case of method name collision in a single inheritance hierarchy, the definition in the most derived class is selected according to the referenced type. For a multiple inheritance hierarchy either the language assumes a precedence ordering or the programmer is called on to disambiguate each method reference. Redefining a method in a derived class is called *method overriding*. As an example, if an object of a derived class has overridden a base class method and we invoke the method we typically want the version defined in the derived class. This can be arranged through dynamic binding of method calls and, in C++, is achieved through *virtual functions*.

A library can be viewed as an object (or collection of objects) by recognizing that during execution there exists some state and some set of operations on that state. The analogy is clear since libraries typically have local data and provide a collection of functions. Libraries are combined in much the same fashion as inheritance by selecting the objects of interest and merging them in a common scope. The result of merging libraries is a new executable image whose exported interface represents the union of merged library component interfaces. Sophisticated linkers allow method overriding, renaming, and hiding, much as in modern object-oriented languages.

One important difference between merging classes using inheritance and merging executable code modules into a program is the linking requirements of code modules. A sophisticated linker might be required to deal with dozens of attributes when building executables. For instance, we would like to reuse existing libraries of non-position independent code (PIC), along with PIC code, or save the results of the dynamic resolution of a program linked with shared libraries. Of course, we would like to avoid recomputing offsets or relocation information whenever possible. These details of linking constraints and code module characteristics could be collected together into an object which can be directly manipulated by an object server. If the executables themselves are objects, then these object descriptor objects are *meta-objects*. The use of meta-objects within OMOS to describe and construct object instances for clients will be discussed in detail in Section 3.

2.1 Module operations

An object server seeking to provide all the features of existing linkers while extending those features into an object-oriented framework must support the flexible combination of modules. Inheritance, as formulated in particular programming languages, is inappropriate as a basis for module combination because it is too burdened with linguistic constraints not directly related to modules. Fortunately, recent work by Bracha[6, 7] and others [8, 15, 22, 29, 17, 9] has centered on formulating a theoretical basis for module combination and manipulation independent of inheritance. The work assumes that we have decoupled inheritance from subtyping and focuses on decomposing inheritance into more basic module operators. These operators are well defined and can be mapped onto languages not directly supporting inheritance or flexible module combination.

In this new treatment of module combination, modules are regarded as mutually recursive scopes which form a uniform space upon which operators (module combinators) act. Module operators accept modules as arguments and return modules as results. In this view a module is much like an abstract data type or object supporting these operations. Modules may include both declarations and definitions of symbols. A declaration gives the type of a symbol but no value binding for it. The set of module operators, as described by Bracha[7], is small and well defined[3]:

Merge: $\|_g$. $m_1 \|_g m_2$ yields the concatenation of m_1 and m_2. The modules must not have any names in common.

Restrict: $\backslash_g$. $m \backslash_g a$ removes the attribute named a from m. If a is not defined, $m \backslash_g a = m$.

Project: π_g. $m \pi_g A$ projects the module m on the names in A. The names in A must be defined in m.

Select: $._g$. $m ._g a$ returns the value of the attribute named a in m. The name a must be defined in m.

Override: $\leftarrow_g$. $m_1 \leftarrow_g m_2$ produces a result that has all the attributes of m_1 and m_2. If m_1 and m_2 have names in common, the result takes its values for the common names from m_2.

Rename: $[\_ \leftarrow \_]_g$. $m[a \leftarrow b]_g$ renames the attribute named a to b. The name a must be defined in m, and b must not.

Freeze: m *freeze* a accepts a module and an attribute and produces a module in which all references to a are statically bound.

Freeze_all_except: m *freeze_all_except* a is the dual operation to freeze.

Hide: m *hide* a performs a freeze on the symbol, then removes the declaration of the symbol from the interface.

Show: m *show* a is the dual of hide and hides all but a in m.

Copy-as: m *copy* a *as* b creates a copy of the a definition under the symbol b.

The merge operator, a binary operator joining the symbols in one module with the symbols in another, subsumes simple concatenation of interfaces and forms the basis of standard linking. When combining modules name conflicts are not allowed and produce an error when encountered. Other operators in the Bracha suite are used to resolve name conflicts.

[3] In [7], Bracha develops a denotational semantics for module operations based on the lambda calculus. This formal semantics represents modules as *generators*, which are functions with a *self* parameter which becomes bound upon module instantiation. Hence, we use a subscript, g, on module operator names.

The restrict operator is used to eliminate the definition of a symbol in a module. The symbol still has a declaration and ultimately requires resolution, but the resolution must come from some other module. In a language like C++ this is analogous to making a method *pure virtual*. (A pure virtual method is a method for which there exists a declaration, but no definition. Classes with pure virtual methods cannot be instantiated, but are very useful for standardizing interfaces to derived classes.) Applying this operator to a library would remove the definition of a symbol and unbind any bound references to it. The project operator is the dual of restrict. Instead of indicating which symbol to remove we indicate which symbols to retain.

The select operator simply returns the value associated with a symbol. (As defined, this operator is not a proper module operator since it returns a value, but is useful in an actual implementation.)

The override operator is a prioritized version of merge. It concatenates its arguments, and in the event of a name collision, selects the definition in the second module. This operator forms the basis for traditional inheritance in object-oriented languages.

The rename operator is a mechanism for resolving name conflicts. If a name a is renamed to b in a module it is as if all textual occurrences of a in the module are changed to b. This allows modules with conflicting names to be merged without loss of functionality.

The freeze and freeze_all_except operations are used to statically bind the definition of a symbol to its use in a module. This fixes its implementation making it "safe" to combine with other modules. In C++ terminology, this corresponds to making a virtual function non-virtual.

The hide and show operations take freeze and freeze_all_except one step further. While freeze removes all references to a symbol (by fixing them) it does not remove the declaration of the symbol from the module interface; hide and show do remove the symbol declaration. The corresponding C++ operation would involve changing a public virtual function into a private non-virtual function.

Finally, copy-as uses the select and merge operations to copy the definition of a symbol and rename it. This is useful when implementing wrappers with modules. If we wish to wrap, e.g., `printf` we cannot use rename to preserve the original function since all references to that function will be renamed. Instead we copy `printf` to a new symbol name, leaving references intact, and substitute our own `printf` which invokes the copied version.

These operators constitute a complete spectrum of operations which are typically bundled into inheritance in object-oriented languages. By decomposing inheritance and making each operation distinct we reduce complexity and enhance flexibility. We use modified versions of these operators in OMOS.

3 Server overview

The object/meta-object server is a repository of objects and meta-objects. Under the OMOS view, objects may be incomplete combinations of code and data fragments, or complete programs. OMOS objects all export and import sets of attributes and attribute values[4]. Conceptually, these objects are treated as modules under OMOS.

Meta-objects contain state and export methods used to construct objects. Thus a meta-object, in effect, contains a class declaration for the object it describes. The class declaration embodied within an OMOS meta-object is interpreted at run-time. These are similar to the first-class objects used to represent classes within languages such as Smalltalk[13] and CLOS[28].

OMOS uses module operations as a mechanism for implementing class hierarchies. While elegant from a language perspective, this framework also provides the building blocks with which one can construct complex object-based systems. A fundamental requirement of OMOS is that it fit well within the traditional framework of programs and libraries, since there exists within this framework a large body of working code we wish to reuse. The module operations OMOS supports allow the interfaces exported by programs and libraries to be redefined, refined, and modified to be more suitable for use within an object-oriented context. Thus, OMOS can reasonably make use of the large amounts of system code that currently exists in the form of programs and libraries. The fact that this code was not necessarily written within an object-oriented context presents little in the way of new requirements for our system; OMOS can project an object-oriented (modular) framework onto a large collection of independently developed programs and libraries.

OMOS clients request object instantiation through a remote procedure call interface. As a result of a request, the `construct` method of a meta-object is executed producing an object. The client is then given a handle through which it can invoke methods on the object. The actual code and data of the instance typically reside within the client address space.

OMOS permits clients to create their own meta-objects, as well as to request the creation of objects from a given meta-object. Since the invocation mechanism is not lightweight, it is expected most OMOS meta-objects will specify medium and heavy-weight objects. Thus we do not envision OMOS as an object construction server in the customary object-oriented run-time system sense. OMOS is designed to support clients running on microkernels such as Mach[1] or Chorus[25], or on traditional monolithic kernels that have adequate VM and IPC facilities.

3.1 Meta-objects

The primary function of OMOS is to produce *instances* of the classes specified by meta-objects. OMOS meta-objects export a `construct` method which can produce instances of the class. Instantiation proceeds by first generating an executable graph of module operations (known as an *m-graph*) through the `decompose` method. The execution of this m-graph produces an object; the object consists of executable code and data fragments (i.e., modules). Full instantiation involves generating an m-graph, executing it to produce a set of modules, assigning address values to the names within a module, then mapping or writing the result into a target address space. M-graphs are a flexible representation of the object; using a graph permits manipulation of the internal structure of the object and its unbound symbols. For instance, they may be used to produce modified versions of the object in other contexts. Meta-objects may cache intermediate results to avoid unnecessary recalculations.

The target address space is typically the address space of the client that instantiates the object. It could also be the address space of a third party to which the client has been given appropriate access.

3.2 Dynamic meta-object modification

As well as using other meta-objects "as is," a meta-object may choose to create an enhanced version of one of its operand meta-objects as part of the instantiation process. It can do so by invoking the `decompose` method of the operand meta-object and modifying the resulting m-graph to produce new behaviors within the operand.

For example, if we wish to extend a meta-object to automatically collect run-time execution profile infor-

[4] The range of values an attribute can have is currently quite limited, since most executable code formats allow only integer values to be assigned to a given symbol. In a more complete implementation attributes would also have an associated type.

mation by linking in a special set of libraries, we derive a profiling meta-object from the original meta-object. The `construct` method of the profiling meta-object first invokes `decompose` on the original meta-object, obtaining the graph of module operations associated with the original object. Next, `construct` traverses the m-graph and replaces each library operand with an alternate profiled version. Then, it executes the modified m-graph to produce a profiled instance of the original object.

Obviously, many other forms of program transformation can be performed by meta-objects in response to various events or conditions. Once profiling information has been collected, it might be used to automatically produce an improved version of the object[23]. Another use of dynamic meta-object modification might be to collect objects into sharable groups. Clustering and sharing related objects have well-known benefits in savings of physical resources, as in the case of shared libraries[26]. Dynamic class modification permits the association of objects based on dynamic and possibly changing requirements.

3.3 Adaptability and extensibility

OMOS is an active entity — i.e., a program with active threads, rather than a static container, such as a file — hence, it can perform its functions dynamically, *adapting* to environmental conditions or explicit client requests. For example, since OMOS returns executable code and data, it can modify the information returned according to the architecture of the system on which the target process resides. The server may adapt its behavior according to the constraints imposed by the target address space.

OMOS itself is constructed in an object-oriented fashion and is extensible. It is implemented from a set of classes which can be extended with dynamic loading. For example, the server can dynamically load alternate classes to process foreign executable file formats. In this fashion, the server can process a wide variety of file formats, without requiring that all formats be built into the server at the time it is constructed.

4 OMOS architecture

OMOS consists of a set of persistent entities which provide basic naming, class construction, and instantiation services. The principal entities of OMOS are:

Directories: organize OMOS components into a hierarchical tree structure;

Meta-objects: describe classes and export methods which are used to produce objects of the class;

Fragments: contain the executable code and data that make up an instantiated object.

4.1 Server object naming

OMOS defines a naming scheme which it uses internally. Clients also use this naming scheme to identify objects of interest. Object names are hierarchical, corresponding to the server directory structure and consist of a series of *name components*. Server directories have a `lookup` operation, which maps a name component to a directory entry (another entity). OMOS separates name components by a slash character ("/"), adopting the same convention as the Unix file system. The `resolve` operation converts a multi-component path name into the object represented by the path.

4.2 Meta-objects

Meta-objects are the basic unit which OMOS uses to describe programs and their construction. Meta-objects define three components: a descriptor for the class they represent; the `decompose` method which generates the m-graph of module operations; and the `construct` method which produces mappable fragments.

4.3 Fragments

Fragments are leaf nodes of the m-graph which contain executable code and data. Fragments export and import interfaces by means of a symbol table. Fragment symbols may be bound to a value or may be unbound. (Depending on the available tools, fragments may be constructed in position independent form, rendering some aspects of symbol/value binding trivial.)

4.4 OMOS operations

Most operations defined within OMOS are analogous to the pure module operations described previously:

Merge: binds the symbol definitions found in one operand to the references found in another. Multiple definitions of a symbol constitutes an error.

Override: merges two operands, resolving conflicting bindings (multiple definitions) in favor of the second operand.

Hide: removes a given set of symbol definitions from the operand symbol table, fixing any internal references to the symbol in the process.

Show: hides all but a given set of symbol definitions.

Rename: systematically changes names in the operand symbol table. The rename module operation can optionally work on either symbol references, symbol definitions, or both.

Copy: duplicates a symbol. The new symbol has the original binding under a different name.

Restrict: erases the definition of a symbol and adds the symbol as a *pure virtual* to the object interface of the class.

Project: restricts all but a given set of symbols.

List: associates two or more objects in a list.

Constrain: constrains the virtual address ranges the operand(s) may occupy.

Annotate: prints an informational message.

Source: produces a module from a source object.

Most operations work on a variable length list of operands. Operands are references to other nodes. Operand nodes may be module operations, meta-objects, or fragments. Upon execution, the majority of these operations produce new modules.

The constrain operation forces its operand to reside within given address space constraints. This operation can *anchor* its operand to a specific address, or restrict it within a range. The constrain operation may be overridden or refined by an enclosing constrain operation.

Constrain operations are used to prevent code from being placed at the same location as existing code, and to allow library designers to segregate groups of objects they wish to share among diverse programs. The same physical copies of the read-only portions of objects may be used by different programs if they share the same <symbol,value> bindings. The address constraints will encourage different clients to make the same bindings. Since the constraints may change and be recalculated dynamically, this scheme does not have the traditional problems of inflexibility associated with binding shared objects to fixed addresses[5].

The source operation invokes a language translator to convert a C++, C, or assembly source file into a fragment. The source file may be provided by a user or dynamically generated within OMOS.

4.5 User interface

OMOS exports an interface to clients that permits them to instantiate objects and to create meta-objects and fragments. To instantiate an object, the client invokes OMOS through a remote procedure call[4] and presents the path name of a meta-object which is to be executed. The user also provides a list of memory regions (a memory constraint) specifying where the resulting object should be placed. A zero-length constraint vector means the object may be placed anywhere. The server returns a list of memory regions occupied by the object and a handle to it (currently, pointers to the entry point and translation table vectors). To construct a meta-object, the user provides a meta-object specification and a server path name to which it is to be bound. To construct a fragment the user specifies the file system path name of a relocatable executable and the server path name to which it is to be bound.

We plan to support remote invocation of methods on objects via an RPC mechanism. OMOS will establish a communication channel between the remote and client processes. On instantiation, the capability for a communication channel will be inserted in the target process by OMOS. OMOS will start a thread in the remote process to service invocation requests; the other end of the communication capability will be returned in the client handle. The communication channel will be used to transmit remote method invocation requests, arguments, and return values between the client and remote tasks.

5 Implementation

OMOS is implemented as a set of C++ classes. Each of the entities described in Section 4 is represented by a class. These classes are made persistent through a set of derived classes (one for each base

[5] If the objects are compiled using position-independent code, there is an implicit level of indirection in <symbol,value> bindings, which eliminates the possibility that a given object could produce multiple bindings; use of PIC simplifies this problem at some performance cost and is in no way precluded.

/template/lib/libc-io:

```
(constrain '(> 0x60000000)
    (show "_open _close _read _write _ioctl"
        (merge /ro/lib/libc/open.o
               /ro/lib/libc/close.o
               /ro/lib/libc/read.o
               /ro/lib/libc/write.o
               /ro/lib/libc/ioctl.o)))
```

Figure 1: Example Blueprint Language

class) that is capable of saving instances on stable storage. Class instances (or server objects) are generally organized in trees, with active portions residing in OMOS memory. References to server objects are controlled and server objects are deleted when no longer referenced.

5.1 Meta-objects

Internally, a meta-object maintains a *blueprint*, which is a program that describes how to construct an instance of the object. The `construct` method for instantiating objects has several stages: the `decompose` method produces the m-graph; the `eval` method is then executed on the m-graph producing a list of fragments; the `fix` method binds symbols to addresses using the address space constraints as a guide; finally the fragments are mapped into the client address space. The m-graph is based on the contents of a *blueprint*. A blueprint is a interpretable representation of a program describing the operations necessary to produce an OMOS module. The `construct` method may be overridden to permit dynamic meta-object modification, as described in Section 3.2.

5.2 Blueprint language

Within OMOS meta-objects, the actual description of how to construct an object instance is encoded in a blueprint using a simple language. The blueprint language uses a simple LISP-like syntax[28]. The language includes operations corresponding to each of the module operations described above. Each operation takes a variable-sized list of arguments; arguments are object names, strings, or other operation expressions. Each operation produces an object as its output.

In the example shown in Figure 1, a subset of the C library is constructed, which exports 5 entry points and is constrained to link at an address somewhere above 60000000 hex. The path name at the top of the figure is used by OMOS to identify the blueprint.

5.3 Module operations

The `compile` method of the blueprint class translates a text representation of a blueprint program into an m-graph. The compilation process resolves names of server objects to references to those objects. Invocation of the `eval` method on the m-graph results in a list of fragments. Each operation recursively `eval`s its operands, then performs its own function on the instantiated operands, returning the result.

5.3.1 Symbol modification operations

A number of operations result in the modification of <symbol,value> bindings within a fragment. These operations use Unix regular expressions to select or modify symbol names. Some operations take as an argument a further specification of the symbol usage (i.e., "reference" or "definition"). Symbol operations result in new objects, although symbol objects that provide different views of the same underlying object share references to a single underlying fragment.

5.3.2 Memory constraint operations

Memory constraints are used to restrict where objects are placed in memory. Memory constraints are sets of <address,size> pairs or specific addresses taken from the domain of the machine address space. Specific addresses are used when an object must start at an exact location; their use is discouraged. Memory constraints may be combined via intersection, union, and complementation. The constraint on a given operand is the intersection of its constraints and all other enclosing constraint operations.

5.4 Class construction via meta-objects

The classes OMOS exports are represented by OMOS meta-objects. OMOS module operations allow combination and inheritance from other OMOS objects and meta-objects.

The graph operations merge, hide, show, override, copy, and rename are composed to implement inheritance between modules. For example, a class may be formed by combining two fragments. If we take the second class to be the base class, we would first show all exported symbols, eliminating extraneous internal symbols. We would then copy all symbols we wish to

/template/lib/libc-debug:

```
(hide "libc%.*"
 (merge
  (rename "reference" "_\(.*\)" "libc%\1"
      /ro/lib/libc/libc-debug.o)
  (rename "definition"  "_\(.*\)" "libc%\1"
      /ro/lib/libc/libc.o))
```

Figure 2: Debugging Interposition Example

be able to continue to access explicitly after inheritance to new, module-specific names (as in packages in LISP). Next, we would override definitions from the base class with definitions from the derived class, so that conflicting definitions are resolved in favor of the derived class, producing a class that represents the combination of the two.

In the current version of the system, inheritance is restricted to the manipulation of object methods, and all data is treated as either global or static object data. The information regarding slot accesses is not available within fragments. If slot offsets are externalized and segregated (so as to be distinguishable from static data), the same module operations apply equally well to member data. We plan, through minor modifications to standard compilation tools, to allow dynamic combination of C++ object data via module operations.

All module operations generate a vector of method addresses, which represents the total set of entry points into the module. The address of this vector is returned to the user on instantiation, along with a table mapping method names to vector indices.

5.5 Interposition

Module operations can easily be used for interposing new routines within an executable. By invoking rename on all definitions of a given set of symbols using some well-known scheme (e.g., prepending a package name), then using rename to change the name of any references destined for the original definition, new values for the symbols in question can be inserted transparently in the original application.

For example, in Figure 2, we produce a version of the C library, libc, where a debugging version of each routine has been inserted to trap calls to the original routine. References to native libc routines in the debugging routine are preserved.

6 Related work

Many other systems exist that support dynamic creation of objects and invocation of methods on those objects. The majority of the systems, such as Argus[19], Eden[2], COOL[14], CLOUDS[10], SOS[27], and COMANDOS/Guide[3] provide a more comprehensive object model which dictate how object distribution and migration are to be accomplished. They tend to use large-grained, active objects. Emerald[5] provides a language model for both active and passive objects. Argus places a special focus on reliability, providing transactional control of operations on objects. CLOUDS has further refined the notion of locality and defined its own extensions to popular languages (notably Eiffel[20], C++, and LISP), segregating its objects into those used *locally* and those used *remotely*.

Relative to complete systems such as these, OMOS provides simple, basic technology. OMOS concentrates on making existing objects available, and allowing extensive combinations of existing objects. OMOS does not seek to provide an all-encompassing object model, but rather is intended to be a useful framework in which one might be implemented. Furthermore, OMOS is oriented towards integration with existing operating system environments rather than rebuilding the foundations of program structure.

Towards the other end of the spectrum, there are a number of interesting shared library implementations[12] which allow multiple clients to share code and data. Most of these facilities are based on a pragmatic, traditional view of programs, and do not provide the ability to dynamically load or recombine objects. Packages also exist to aid programmers in the dynamic loading of code and data[16]. These packages tend to have a procedural point of view, and provide lower-level functionality than OMOS.

7 The results

OMOS is in experimental use as an object server running on top of the Mach operating system, acting as an object and program repository and providing, indirectly, a shared library service. The majority of module operations have been implemented. A port to BSD Unix[18] is planned. OMOS has been used to conduct experiments in automatically generating locality-of-reference optimization in running systems[23]. The basic OMOS system comprises 7,100 lines of C++ code.

Tests show OMOS to be efficient. OMOS meta-objects cache intermediate results, allowing them to avoid unnecessary recalculations. The Unix `exec` system call, when implemented using OMOS facilities, runs 33% faster than the standard Unix `exec()` system call on a 3.0 Mach system running the monolithic BSD Unix server. A benchmark program which `execs` itself 5000 times, executes in 69 seconds using the standard Unix `exec`. The same program executes in 46 seconds using the OMOS `exec` facilities.

8 Conclusion

In this paper we described the OMOS system, an object-oriented approach to program binding and execution. This system is an attempt to apply the concepts of object-oriented programming languages along with new work in modularity and inheritance to the problem of traditional program composition. One important goal of the work is that the current investment in compilation, linking, and execution technology not be lost as we move towards a more comprehensive framework. As such, this work provides a incremental migration path towards a more object-oriented environment with incremental payoff.

A prototype OMOS system is currently running and has proven to be flexible enough to provide a traditional high quality program execution and shared library service while, at the same time, maintaining an object-oriented framework. Module operations have proven to be powerful enough to use as the basis for class construction, and their successful use has opened a number of promising avenues in the fields of language design and object-oriented programming.

In further work with OMOS we plan to focus on more non-traditional aspects of program construction, concentrating on dynamic interposition of modules, a more distributed implementation, and support for alternate implementations of modules to provide enhanced functionality.

Acknowledgements

We would like to thank Gary Lindstrom and Gilad Bracha for the crucial insights and considerable time spent reviewing this work. In addition, we would like to thank Bob Kessler and Jay Lepreau who provided valuable comments on early drafts of this paper.

References

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, Atlanta, GA, June 9–13, 1986. Usenix Association.

[2] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe. The Eden system: A technical review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59, January 1985.

[3] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. Le Dot, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scoiville, and G. Vandôme. Architecture and implementation of Guide, an object-oriented distributed system. *Computing Systems*, 4(1):31–67, Winter 1991.

[4] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1), February 1984.

[5] A. P. Black, N. Huchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Trans on Software Engineering*, SE-13(1):65–76, 1987.

[6] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, March 1992. 143 pp.

[7] 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.

[8] L. Cardelli and J. C. Mitchell. Operations on records. Technical Report Tech. Rep. 48, Digital Equipment Corporation Systems Research Center, August 1989.

[9] W. Cook, W. Hill, and P. Canning. Inheritancee is not subtyping. In *Proceeding of the ACM Symposium on Principles of Programming Languages*, pages 125–135, 1990.

[10] Partha Dasgupta, R. Ananthanarayanan, Sathis Menon, Ajay Mohindra, Mark Pearson, Raymond Chen, and Christoper Wilkenloh. Language and

operating system support for distributed programming in Clouds. In *Proceedings of the Symposium on Experiences with Distributed and Multiprocesor Systems (SEDMS II)*, pages 321–340, Atlanta, GA, March 21–22, 1990. Usenix Association.

[11] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA, 1990.

[12] R. A. Gingell, M. Lee, X. T. Dang, and M. S. Weeks. Shared libraries in SunOS. In *Usenix Conference Proceedings*, pages 131–145, Phoenix, AZ, Summer 1987. USENIX.

[13] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[14] Sabine Habert, Laurence Mosseri, and Vadim Abrossimov. COOL: Kernel support for object-oriented environments. *ACM SIGPLAN Notices*, 26(4):269–277, October 1990.

[15] R. Harper and B. Pierce. A record calculus based on symmetric concatenation. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 131–142, January 1991.

[16] W. Ho and Wilson. *DLD: A Dynamic Link/Unlink Editor.* Free Software Foundation.

[17] R. E. Johnson and V. F. Russo. Reusing object-oriented designs. Technical Report UIUCDCS 91–1696, University of Illinois at Urbana-Champagne, May 1991.

[18] W. N. Joy, R. S. Fabry, S. J. Leffler, and M. K. McKusick. *4.2 BSD System Manual.* Computer Systems Research Group, Computer Science Division, University of California, Berkeley, CA, 1983.

[19] Barbara H. Liskov. *Distributed Systems: Methods and Tools for Specifications*, chapter The Argus Language and System, pages 343–430. Lecture Notes in Computer Science no. 190. Springer-Verlag, 1985.

[20] B. Meyer. Eiffel: Programming for reusability and extendability. *SIGPLAN Notices*, 22(2), February 187.

[21] Bertrand Meyer. *Object-oriented Software Construction.* Prentice-Hall International, Hertforshire, England, 1988.

[22] J. Mitchell, S. Meldal, and N. Madhav. An extension of standard ML modules with subtyping and inheritance. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 270–278, January 1991.

[23] Douglas B. Orr, Robert W. Mecklenburg, Pete Hoogenboom, and Jay Lepreau. Dynamic program monitoring and transformation using the OMOS object server. Submitted for publication.

[24] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 57(6):1905–1930, July/August 1978.

[25] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. The Chorus distributed operating system. *Computing Systems*, 1(4):287–338, December 1989.

[26] Donn Seeley. Shared libraries as objects. In *Proceedings of the Summer 1990 USENIX Conference*, pages 1–12, Anaheim, California, June 11–15, 1990. Usenix Association.

[27] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An object-oriented operating systems—assessment and perspectives. *Computing Systems*, 2(4):287–337, Fall 1989.

[28] Guy L. Steele Jr. *Common Lisp: The Language.* Digital Press, second edition, 1990.

[29] Alan Synder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 38–45, 1986.