ETYMA: A Framework for Modular Systems*

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

Abstract

Modularity, i.e. support for the flexible construction, adaptation, and combination of units of software, is an important goal in many systems. In most cases, however, systems achieve only a few aspects of modularity. The problem can be traced to the inflexibility, or the limited view of modularity taken by the underlying architecture of these systems. As a remedy, we show that the notions fundamental to object-oriented programming, i.e. classes and inheritance, can be formulated as a simple meta-level architecture that can be effectively reused in a wide variety of contexts. We have realized such an architecture as an O-O framework, and constructed two significant and distinct completions of it. Systems based on this framework benefit not only from design and code reuse, but also from the flexibility that the architecture offers. In addition, the architecture represents a unification of the fundamental ideas of several similar but subtly different module systems.

*This research was sponsored 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 and by the Department of the Army under Grant number DABT63-94-C-0058. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

†Primary contact author. Phone +1-801-581-8378, fax +1-801-581-5843.

1 Introduction

It is widely agreed that software design reuse is beneficial. Functional factoring and interface design are indeed difficult tasks, and it is worthwhile to reuse the fruits of these activities. Object oriented programming helps us achieve design reuse through inheritance and polymorphism.

The basic thrust of this paper is that the ideas underlying object-oriented programming are themselves worthy of reuse. O-O classes and inheritance appear in many forms in various languages and systems. Throughout, their basic goal is the same: to support the construction, adaptation, and combination of units of software. We claim, therefore, that a suitably designed model of classes, one that is malleable enough to express various forms of combination and adaptation, can be reused in many contexts, including some that are not usually thought of as object-oriented programming.

We believe we have designed such a model, based on an austere notion of classes coupled with a powerful suite of operators to manipulate them, borrowed from a module manipulation language called *Jigsaw*[BL92]. We have designed an O-O framework called ETYMA that incorporates this notion of classes as well as abstractions covering much of the value space and type space commonly found in module-based languages. Besides the concepts of classes and their instances, ETYMA models records, functions, primitive values, variables (locations), and their corresponding types. The high-level design of ETYMA is simple, yet completions of this framework are surprisingly powerful in their ability to manipulate classes.

ETYMA is a meta-level language architecture similar to those of familiar languages such as Smalltalk and CLOS MOP. It has the same advantages of enabling the construction of reflective, flexible, and extensible programming systems[BL94]. However, the differences are crucial. Firstly, and most importantly, classes and inheritance in most systems are composite notions serving many purposes. In ETYMA, classes are very simple units of software that can be composed using a powerful set of operators. The merit of ETYMA arises primarily from the fact that this architecture can be reused outside the realm of traditional O-O programming. For example, we have reused it in the design of a programmable linker/loader, which we describe in Section 4.

Secondly, ETYMA can not only be used to construct processors for dynamic reflective languages, but also for compiled languages, since it supports static typechecking and separate compilation. A compiler can use the abstractions in ETYMA for representing the semantics of language constructs in an intermediate form. Such compiler frameworks can augment the evolvability and maintainability of compilers written using them.

Thirdly, by virtue of its structuring, ETYMA can be used to layer object systems on top of existing, even non O-O, languages. This is in addition to its more obvious use for building a processor for a new language. We describe the design of an object system layered on top of ANSI

C in Section 4, as well as an extended CORBA Interface Definition Language [COR91] in Section 5.

In Section 2, we introduce our model of classes and inheritance and describe its realization in the design of the abstract and concrete classes of the ETYMA framework in Section 3. We then examine our linker completion in Section 4, and outline our IDL completion in Section 5. Finally, we sketch related and future work and conclude.

2 Model of Classes and Inheritance

The concepts outlined in this section, drawn from the module manipulation language *Jigsaw*, provide the semantic basis for the design of our meta-level language architecture.

As mentioned, current O-O languages embody varying notions of the *class* concept, each of which differs from others in subtle but important ways. These different notions share a common semantic goal: to facilitate the structuring and combination of software units with well-defined interfaces. We use the term *module* to refer to such software units. In *Jigsaw*, a module is a self-referential scope, consisting of a set of definitions and declarations. Definitions bind labels (identifiers) to typed values, and declarations simply associate labels with types (defining a label subsumes declaring it). Declarations are used to create abstract modules, which can be manipulated but not instantiated. Modules do not contain any free references, i.e. references to labels that are not associated with any declarations either locally or in some surrounding scope. Every module has an associated interface, which comprises the labels and types of all its visible attributes. Interfaces are purely structural, i.e. sets of label-type pairs, without order or type name significance.

Traditionally, classes fulfill a variety of roles, including defining modules, defining subtyping relationships, controlling visibility (e.g. via public/protected/private interfaces), constructing instances of a defined module, modifying and reusing existing program units via single inheritance, combining program units using multiple inheritance, resolving name conflicts, etc. In *Jigsaw*, such effects are made possible via a suite of module operators (*combinators*), each designed to fulfill a single isolatable semantic role. The primary operators are merge, override, restrict, rename, freeze, hide, and copy-as.

To illustrate the use of some of these operators, consider the example in Figure 1. Suppose in module M, we wish all invocations of the function f to be redirected to a new definition of f using both its own definition, which it refers to as f, and the existing definition of f, which it refers to as f_old. For this, we first copy attribute f as f_old in module M and override f with the new definition in module IM. The operator override enables module combination with conflicts resolved in favor of the right operand. We then encapsulate the old definition f_old, using the operator hide. In this

(1) Original module

```
M = module
   g = fun() ...f(2)... end;
   f = fun(y:int) returns int
         if y > 1 then 1+f(y-1) else 3; end;
end;
```

(2) Interposing function module

```
IM = module
   f_old : int -> int;
   f = fun(y:int) returns int
    if y > 10 then f(10) else f_old(y); end;
end;
```

(3) Copy M's f as f_old (M1 has 3 attr's: g, f, f_old)	`M1 = M copy_as f f_old;`
(4) Override M1's f with IM's f	`M2 = M1 override IM;`
(5) Hide M2's f_old (M_result has M's interface)	`M_result = M2 hide f_old;`

Figure 1: Function interposition via *Jigsaw* operations.

manner, composite modules can be constructed from simple ones by performing a series of module transformations. The module operators and their semantics will become clearer as we progress through the paper.

Bracha and Lindstrom [BL92, Bra92] have given a rigorous formal semantics for *Jigsaw*'s module manipulation, building on the work of Cardelli, Cook, and others [HP91, CM89, CP89, BC90]. They have formulated *Jigsaw* in such a way that it does not prescribe the computational domain, or the control structures, or even the surface syntax of the concrete language in which it is used. This abstract formulation has facilitated its realization as an O-O framework.

3 The Design of ETYMA

In order to broadly and usefully apply a generalized view of O-O programming such as the above, one needs a practical, coherent, and reasonably complete realization of it. For this, we exploit the idea that generic linguistic notions such as "module," "record," "instance," etc. can be organized into a taxonomy of concepts with relationships such as IS-A, HAS-A, AGGREGATES etc. Furthermore, such a space of concepts can itself be specified using an O-O language, thus constituting an O-O *framework*. Such a framework then defines a meta-level language architecture applicable to modular systems.

In essence, an O-O framework is an O-O model that captures the essential abstractions in a particular application domain [JR91]. It expresses the architecture of applications in the domain in terms of objects and interactions between them. Frameworks allow developers to build applications effectively by concretizing abstract classes in the framework via inheritance and by configuring, i.e.

connecting instances of, predefined concrete classes in the framework. As a result, a framework can be thought of as being parameterized on a completion that provides *call back* code — a sort of bi-directional function abstraction. Thus, applications are built by *completing* a framework for specific purposes, while preserving the overall architecture of the framework. Frameworks thus promote design and code reuse through O-O concepts such as inheritance and polymorphism.

In this section, we present our framework for modular systems. We call this framework ETYMA (the plural of "etymon," obtained from the etymology of "etymology") since it is a collection of "root" concepts from which other concepts are formed by composition or derivation. Section 3.1 presents the abstract classes in ETYMA, Section 3.2 presents some concrete classes, and Section 3.3 presents type-system related classes. These sections together describe ETYMA. Throughout the paper, we use the diagramming conventions[1] as well as the concept of *design patterns* introduced in [GHJV95] to describe ETYMA.

3.1 Abstract Classes

Figure 2 shows an overview of the abstract classes in the ETYMA framework. Classes *Type* and *TypedValue* are abstract superclasses that model the linguistic domains of types and values respectively. ETYMA models strong typing; hence concrete subclasses of *TypedValue* are expected to return their concrete type object (see Section 3.3) when queried via *type-of()*.

The abstract class *Module*, a *TypedValue*, captures our notion of module in its broadest conception. All operations that can be performed on modules are specified as abstract methods of the class. At this level, the representation of the attributes of modules, as well as the implementation of operations on them, are left unspecified. These are expected to be provided by concrete subclasses.

The semantics of the methods of class *Module*, which are at the heart of this model, are given informally in Table 1. Module operators are applicative, i.e. they always produce new modules without mutation of operand modules.

As mentioned earlier, the methods are understood to be primitive operators which can be composed in various ways to achieve effects such as inheritance, encapsulation, sharing, and nesting. For example, the single inheritance model of instance variables and methods in Smalltalk is similar to *hide*'ing the instance variables followed by *override*'ing the methods in the superclass with those of the subclass. Access to **super** methods is achieved with the application of the *copy-as* operator to superclass methods, as illustrated in Figure 1. Furthermore, static binding of self-references, akin to non-virtual member functions of C++ [ES90], is achieved via the operator *freeze*. Name conflict

[1] Labeled boxes stand for classes; lines with triangles stand for inheritance (IS-A); arrowhead lines indicates object references (HAS-A); lines with diamonds indicate aggregation; slanted font indicates abstract classes and methods; regular font stands for concrete classes; and boxes at the end of dotted lines indicate code fragments.

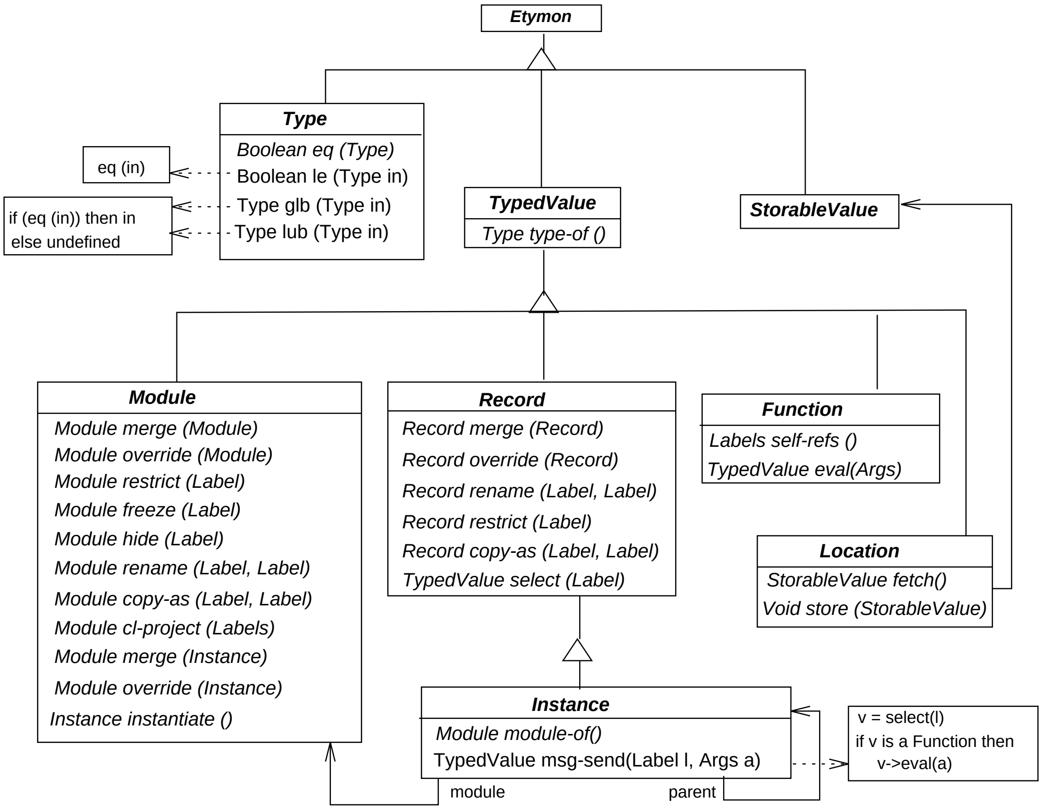

Figure 2: Overview of abstract classes

resolution in the case of multiple inheritance is achieved with the aid of the *rename* operator. Other applications of these operators are given throughout the paper.

The *instantiate()* method of class *Module* is a factory method (pattern) that is used to generate instance objects of module objects. This method returns an object of a concrete subclass of class *Instance*. Class *Instance* is modeled as a subclass of *Record* following the formal semantics of *Jigsaw*. Formally, modules are modeled as record generators of the form $\lambda s.\{a_1 = v_1, ..., a_n = v_n\}$, with the variable s standing for self-reference. The fixpoint of such a generator is a record with its self-references bound. This models instances. Consequently, an instance "IS-A" record.

Class *Record* models the classical notion of records: functions from labels to values, with no self-reference. *Record* supports operations such as merge and restrict, similar to the ones found in [CM89, HP91]. In particular, the method *select(Label)* models attribute selection. Class *Instance* supports operations similar to *Record*. In addition, class *Instance* models the traditional O-O notion of sending a message (dynamic dispatch) to an object as *select*'ing a function-valued attribute followed by invoking *eval* on the returned function object. This functionality is encapsulated by the template method (pattern) *msg-send(Label,Args)*.

Method	Semantics
merge(Module)	combine modules in the absence of label conflicts
override(Module)	combine modules resolving label conflicts in favor of the incoming module
restrict(Label)	strip an attribute of its definition, retaining only its declaration
freeze(Label)	statically bind all references within module to the given attribute
hide(Label)	bind references to attribute and remove it from interface
rename(Label,Label)	rename an attribute and all its uses in the module
copy-as(Label,Label)	copy an attribute definition giving it another Label
cl-project(Labels)	project out given attributes and the closure of their self-references
merge(Instance)	combine module with instance in the absence of label conflicts
override(Instance)	combine module with instance resolving conflicts in favor of incoming instance

Table 1: Semantics of methods of class *Module*

Further, class *Instance* has access to its generating module with the *module-of()* method. An instance of a nested module has access to its surrounding instance object via its parent member.

Mutable state (e.g. instance variables) is modeled with class *Location*. Location objects hold *StorableValue*'s, the exact definition of which depends on a particular completion. For example, storable values are typically at least the primitive values in a language, but often include pointers, which can be modeled as locations containing other locations. Function-values are modeled by class *Function*. The role and use of these abstractions will become clearer with the description of their concrete subclasses in the following section.

3.2 Concrete Classes

As described thus far, the framework provides a rather generic object model, abstracting over notions such as primitive values and control structures in potential language completions. This basic architecture itself can be used for constructing various kinds of modular systems, one of which is described in Section 4. However, in order to be directly useful, e.g. for constructing a *Jigsaw*-based language compiler, concrete subclasses of generic notions must be provided as part of the framework. Figure 3 gives an overview of the important concrete classes, and helps clarify the meta-architecture.

Class StdModule is a concrete subclass of *Module* that represents its attributes as a map (class AttrMap). An attribute map is a collection of individual attributes, each of which maps an object of class Label to one of AttrValue. Attribute maps are used to implement StdModule as well as StdInterface, which represents the type of StdModule objects (see Section 3.3). Hence, AttrValue holds an object of a concrete subclass of either *TypedValue* (definitions) or *Type* (declarations).

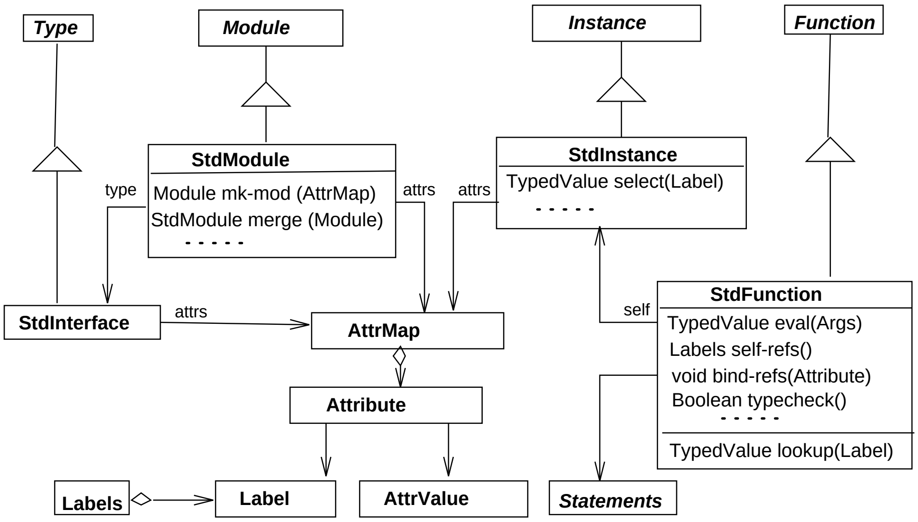

Figure 3: Overview of some concrete classes

Given this implementation of attributes, module combination methods are implemented as transformations on attribute maps. For example, the merge operation concatenates the attribute maps of operand modules if there are no conflicting definitions.

The instantiate method of StdModule objects yield StdInstance objects, which are also implemented using attribute maps. Instances of a module share all the module attributes except location-bound attributes (corresponding to instance variables), which are cloned per-instance at instantiation time. Variable sharing among instances (as with static members of C++) is supported via the use of nested modules. A location in a surrounding scope is shared by all instances of nested modules.

Class StdInstance implements the method select(Label) to perform a simple lookup of a label's binding, possibly in a surrounding instance, and return its value. Attributes in lexically surrounding scopes are accessed via parent. The inherited msg-send method implements the message sending operation as described earlier. Of course, the method can be refined to incorporate alternate, more efficient dispatch mechanisms [Cha89] in other concrete subclasses.

Reference to *self* is an important aspect of O-O programming. Self-reference indirection enables dynamic binding, which in turn enables polymorphism. Self-reference occurs within methods. Methods, i.e. function-valued module attributes, are modeled in ETYMA using StdFunction, a concrete subclass of class *Function*. An object of StdFunction has access to the instance within which it is executing via its member self. The self object is passed in as a parameter to eval. When the function object needs the value of a self-referenced attribute, its lookup method sends the message select(Label) to its self parameter. This corresponds to dynamic binding.

Interestingly, modules in our model require another form of delayed binding occurring not at run-time but rather at module combination time. This is because of the ability to statically bind self-reference via freeze. That is, self-references to module attributes are by default dynamically bound, but can be fixed at module combination time by applying the module operator freeze. Thus, reference to frozen attributes need not be indirected via self dynamically. This type of binding is implemented by the method bind-refs(Attribute) of StdFunction, which copies and stores the attribute in its local environment for subsequent access. The lookup(Label) method of StdFunction fetches locally stored bindings for labels before accessing self.

The hide(Label) method of StdModule implements encapsulation by statically binding accesses to its label parameter using the bind-refs(Attribute) method of StdFunction, and then removing the attribute from its attribute map.

Control structures (statements) that make up function objects are given by concrete subclasses of class *Statement*. Subclasses of *Statement* correspond to the traditional implementation of abstract syntax trees. These are not described further.

The module method cl-project(Labels) implements the functionality of extracting (or "projecting," by analogy to relational calculus) a subset of the attributes of a module. This subset is given by the closure of self-references within the bindings of the given labels. The self-references that occur within function valued bindings are obtained with the method self-refs() of StdFunction.

3.3 The Type Classes

A type system is built into the framework. The ETYMA framework class *Type* has an extensive set of concrete subclasses that model types commonly found in modular programming languages.

As mentioned earlier, the type of a StdModule object is a StdInterface object. Class StdInterface, a concrete subclass of *Type*, implements methods that typecheck individual module operations. Methods of StdModule call methods of class StdInterface such as mergeable(StdInterface), overrideable(StdInterface), etc., which implement the type rules for merge, override, etc. These methods are based on the type rules of the *Jigsaw* language, given informally in Table 2. As can be seen, these type rules depend on notions of type equality and subtyping. In addition, type rules for merge and override need to compute the greatest common subtype of a pair of types.

As a result, class *Type* has an abstract method *eq* that checks if two types are equal. In addition, it has template methods (pattern) le that checks for subtype (defaults to eq) and glb that computes the greatest common subtype. In order to compute the greatest common subtype of two function types, it is required to compute the lowest common supertype of their input argument types, due to contravariance. As a result, class *Type* defines a method lub that computes the least common

Operator	Typing
merge(Module)	Matching definitions disallowed; a definition must be a subtype of matching declaration. If there are matching declarations, then replace with greatest common subtype.
override(Module)	Same as merge, except definition conflicts allowed; incoming definition must be a subtype of matching definition.
restrict(Label)	Label must be defined.
freeze(Label)	Label must be defined.
hide(Label)	Named attribute must be defined.
rename(Label,Label)	First attribute must exist; second must not.
copy-as(Label,Label)	First attribute must be defined; second must not.
cl-project(Labels)	Labels must exist.
merge(Instance)	Same as rule for merge(Module)
override(Instance)	Same as rule for override(Module)

Table 2: Informal type rules for module combination

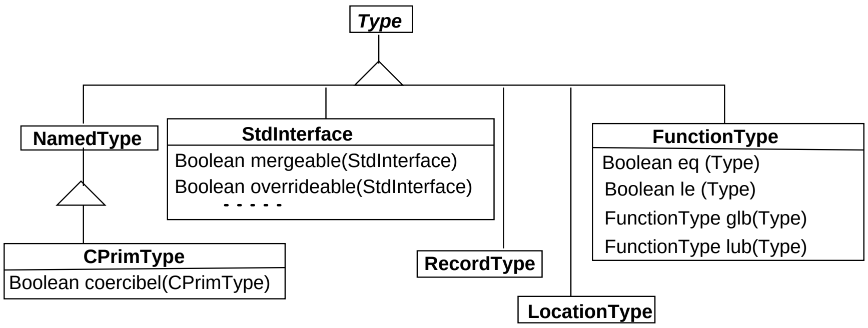

Figure 4: Some concrete type classes

supertype of a pair of types. Concrete subclasses of *Type* are expected to appropriately redefine this semantics.

Figure 4 shows the concrete type classes. Class NamedType models types that have identity. For named types, equality is determined by equality of identities. Subtyping is often given by type equality. An example subclass of NamedType is a class representing the primitive types of C, CPrimType. In addition to its superclass' notion of equality, this class implements C language rules such as short is the same as short int, etc. It could also define coercibility relationships among primitive types, e.g. an int can be coerced into a float, etc.

Class FunctionType models function types with the standard notions of equality and subtyping, taking into account contravariance. Methods to compare function types are used in the combination of modules that contain function-valued attributes. This should be distinguished from typechecking the *implementation* of a function, which is done by calling the typecheck() method of the function object, which typechecks the statements comprising the function.

Location-bound attributes (variables) can be used as evaluators (*i.e.* expressions that return values) and as acceptors (*i.e.* expressions that receive values) in different contexts. Expressions which are evaluators can only be replaced with expressions whose types are subtypes of the original, while expressions which are acceptors can only be replaced by expressions whose types are supertypes of the original [Bru92]. As a result, subtyping of variables is always restricted to type equivalence.

Classes StdInterface and RecordType support structural typing, in which the names and types of attributes, but not their order, is significant for type checking . Furthermore, the module type system separates inheritance from subtyping, i.e. a module operation does not necessarily result in a module that is a subtype of the original module(s). For example, consider the hide operation, which shrinks the interface of a module, and often results in a supertype.

This concludes the description of the ETYMA framework. ETYMA is implemented in C++ (currently about 10K lines). A direct, although cumbersome, way to use this framework is by writing C++ programs that instantiate these classes and use the modularity features. Another way is to write a parser that instantiates classes corresponding to input source with some surface syntax and then generates, say, corresponding C code. We have experimented with both these ways for "little" modular languages. However, we describe more significant completions of ETYMA in Sections 4 and 5. ETYMA has undergone several iterations over the last two years, especially those involving the completions, and has evolved to its current form.

A distinguishing feature of ETYMA is that its design has been guided mainly by a formal description (i.e. denotational semantics and type rules) of the corresponding linguistic concepts. The reader might have noted the correspondence between the above framework abstraction design and denotational models of programming languages [Gor79]. Denotational semantics applies functional programming to abstract over language functionality. Here, we apply a denotational description of modularity to abstract over language modularity. Furthermore, the framework approach is intended to provide the language developer a modular means by which to design *and implement* a language's value domain, type system, etc. relatively independently of each other. Once the basic elements of the language are designed, the classes in the framework are directly available for incorporation into the language processor.

4 A Linker Completion

Object-oriented classes represent logical modularity in programs. In non O-O languages, we can still exploit *physical* modularity and apply the module model described above. For example, source program files in ANSI C can be viewed and manipulated like modules. So can object modules (".o" files, which we will refer to as *dot-o*'s) which are compiled forms of source modules. In fact, dot-o's have a rather simple structure and fit nicely into our notion of modules.

A dot-o generated by compiling an ANSI C source file consists of a set of attributes with no order significance. Here, 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;)[2]. 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) data and functions as encapsulated attributes. Furthermore, a dot-o contains unresolved self-references to attributes, represented in the form of relocation entries.

Analogously, 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 be instantiated multiple times, bound to different addresses. Hence, fixed executables can be modeled as instances of dot-o's.

Obviously, it is highly beneficial to build tools that manipulate dot-o's and fixed executables as Etyma modules and instances. Such tools enable the use of structuring and composition techniques such as O-O inheritance on what are traditionally viewed as rigid system artifacts. We have designed just such a tool, a second-generation programmable linker called OMOS (Object/Meta-Object Server) [OM92], as a completion of Etyma. OMOS is designed as a continuously running server that not only manipulates modules, but also constructs instances and maps them into process address spaces, possibly after performing various optimizations.

This section describes the design of OMOS's module manipulation functionality. The design of object modules (class DotO), fixed executables (class FixedExe), and module combination scripts (class ModuleSpec) are shown in Figure 5, and described further in the following sections.

4.1 Composition of Object Modules

Object modules are represented by class DotO. Most object file formats provide for a symbol table and relocation information along with text (read-only code) and data. The symbol table consists of entries for file-level attributes that are both exported from (definitions) and imported

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

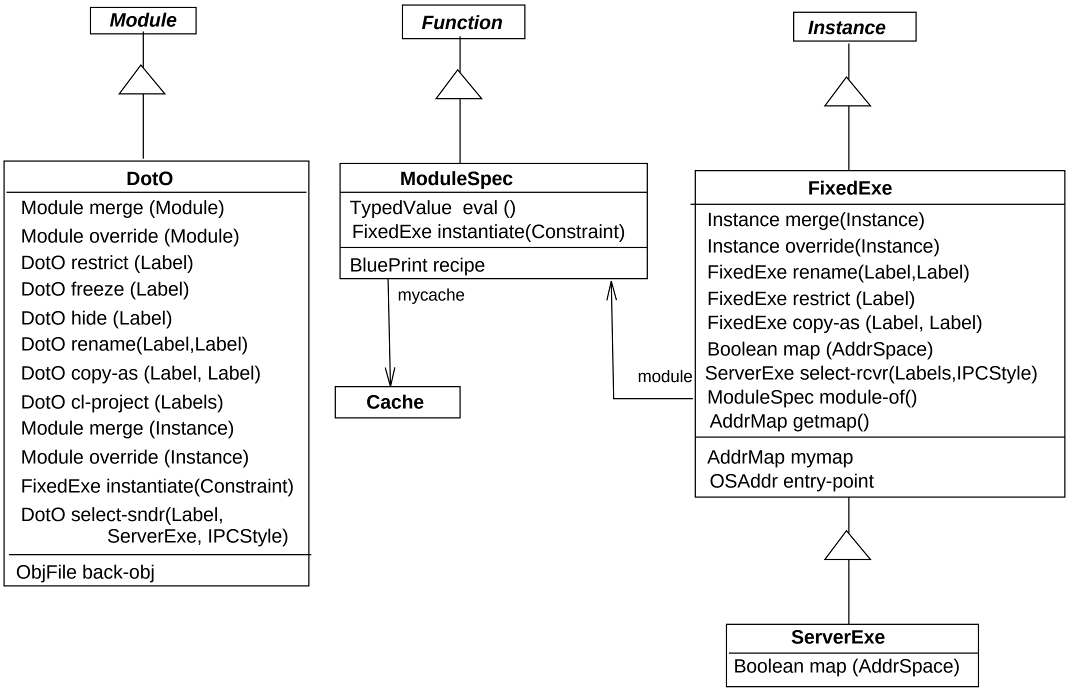

Figure 5: Overview of the OMOS completion

into (declarations) the file. Relocation entries correspond to self-references from methods to file-level attributes. The above two kinds of information are basically sufficient to implement the module operations of ETYMA. Class DotO delegates its operations to class ObjFile which operates on the internal format of dot-o symbol tables and relocation using the BFD package [Cha92].

Most module operations are transformations on the symbol table of the object file. The merge operation on a DotO is equivalent to traditional linking, but without fixing relocations. Declarations are matched up with definitions, and conflicting (multiply defined) symbols are flagged as an error. The restrict operation converts a definition into a declaration (extern, or undefined). The hide operation removes a definition from the external interface of the object file, i.e. makes the definition static. It is in general not possible to perform the freeze operation on individual attributes of a dot-o since freezing an attribute amounts to fixing the address of the attribute, whereas addresses are not known until instantiation time.

Module expressions that use operations such as the above can perform several useful transformations on programs at link-time [OMHL94]. For example, monitoring a program involves transforming the program so that each defined procedure is transparently *wrapped* with an outer routine that monitors entry to and exit from the procedure. The wrapper can be generated automatically. Figure 6 shows the module operations used to wrap the procedure f of module M1 with the automatically generated routine found in M2. Note that this example is similar to the example

```
/* Module M1: */
void g () {
    short z = f (3);
}
short f (short x) {
    /* ... */
}
```

```
/* Module M2: */
/* Automatically generated */
extern short __f (short);
extern void _log_enter (char *);
extern void _log_exit (char *);
short f (short x) {
  short v;
  _log_enter ("f");
  v = __f (x);
  _log_exit ("f");
  return v;
}
```

Module expression: ((((M1 copyas f __f) restrict f) merge M2) hide __f)

Figure 6: Wrapping a routine to monitor its execution

in Figure 1.

Module expressions for useful operations are shown in Table 3, and described further in the following sections.

Operation	Script
Link M1 and M2	(merge M1 M2)
Wrap f in M1	((((M1 copyas f __f) restrict f) merge M2) hide __f)
Archive merge	(merge M (cl-project ARLIB (X Y ...)))
IPC	(select-sndr A (select-rcvr (A B) MACH) MACH)

Table 3: Examples of operations on DotO and FixedExe objects

4.2 Instantiation and message sending

In Etyma, 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 amounts to fixing relocations in the dot-o, and storage allocation amounts to binding addresses. These two steps are usually performed simultaneously.

A FixedExe object is represented as an address map (class AddrMap). 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. The map operation of a fixed executable is responsible for actually mapping the file into the address space of a process, and starting its execution at its entry-point.

The implementation technology of executables matters a lot in the realization of instance operators like merge. For example, it is easier to merge an executable with position independent code (PIC) as compared with one that is not PIC. For non-PIC, merging two fixed executables requires unbinding of addresses if they happen to overlap. There usually isn't enough information in a fixed executable to unbind addresses. In such cases, a new, appropriately bound FixedExe can be generated from its DotO object. On the other hand, PIC code is bound in a relative manner, and data accesses are usually indirect, hence unbinding is not necessary.

It is often necessary to merge a DotO with a FixedExe object, e.g. with a fixed shared library (see Section 4.3). The methods merge(Instance) and override(Instance) methods of DotO support these operations. These methods fix the dot-o at non-overlapping addresses with respect to the executable.

An important issue concerns the meaning of the select(Label) operation of class *Instance* on class FixedExe. As mentioned earlier, the notion of "select" on instances in the framework corresponds closely to message-sending in traditional O-O programming. A fixed executable can be thought of as a persistent version of a program that has been mapped to a process' address space. Hence, for fixed executables, the message-sending operation becomes a form of communication between mapped executables, or a form of inter-process communication (IPC)!

This idea is realized in the OMOS completion with the methods select-rcvr of class FixedExe and method select-sndr of class DotO. The method select-rcvr(Labels,IPCStyle) returns a fixed executable (a ServerExe) which has IPC receiving stubs incorporated into it for particular labels. The method select-sndr(Labels,ServerExe,IPCStyle) returns a dot-o that has IPC calling stubs incorporated for particular *declared* attributes. The map operation of ServerExe is redefined to validate and establish the ServerExe object as a server prepared to accept incoming IPC calls from clients generated by select-sndr. Thus, this technique provides a way by which a regular intra-process static function call can be converted into an inter-process function call via programming at link-time. The particular variety of IPC to be used for generating sending and receiving stubs, e.g. DCE or CORBA, is specified as an extra argument (IPCStyle) to the select methods.

4.3 Modeling Libraries

Traditionally, libraries with various semantics are dealt with by linkers and loaders [Gin89, See90]. Hence, it is necessary for OMOS to model and manipulate libraries.

Archives are a common kind of library, e.g. libc.a on UNIX systems. When a dot-o is linked with an archive, only that part of the archive that is required by the dot-o is extracted from the archive, and linked in. In order to model this semantics in our linker, we use the cl-project operator to extract only those definitions that we need, then merge it with the module, as shown in Table 3.

Shared libraries are special forms of modules and instances. Fixed address shared libraries are constrained to occupy a certain pre-determined area in a process' address space. Programs that need to use a fixed shared library must map themselves into parts of the address space that do not conflict with any library. OMOS supports a mechanism by which programs can specify address constraints for instantiating a dot-o into a fixed-exe. As a consequence, fixed shared libraries are modeled simply as fixed executables in OMOS.

Dynamic shared libraries are not fixed prior to mapping. In fact, such libraries contain declarations that are resolved (linked) only at run-time. Hence, dynamic shared libraries are modeled simply as dot-o's. A variation of the select-sndr operation converts a regular dot-o to a dynamic shared library by merging in the necessary stubs to perform dynamic loading and linking. Such a dot-o is fixed at map-time, and executed. At run-time, when the loading stub is invoked, the necessary libraries are dynamically loaded, and binding performed.

4.4 Module Scripts

OMOS supports a stable store in which object files and persistent versions of executables are bound to names. In addition, the persistent store contains named scripts that specify operations to be performed on other named entities. The scripting language, an extension of Scheme, supports first-class modules, a suite of module and instance operations, including operations for constructing an object module given program source code, and persistence store management functionality.

A named module composition script is a function that returns a DotO or FixedExe object, hence is modeled as a subclass ModuleSpec of *Function*. Note that a ModuleSpec object is not a function-valued attribute of a module, but rather a stand-alone named function.

4.5 OMOS Services

The OMOS linker/loader is designed to provide a dynamic linking and loading facility for client programs via the use of module combination and instantiation. This facility is used as the basis for system program execution and shared libraries[OBLM93].

Since OMOS is an active entity (a server), it is capable of performing sophisticated module manipulations on each instantiation of a module. Evaluation of a ModuleSpec object could potentially produce different results each time. Some OMOS operations such as those used to implement program monitoring and reordering [OMHL94] enact program transformations using operations on module expressions themselves.

Since OMOS is capable of performing potentially complex manipulations on each invocation, it caches the results of most operations to avoid re-doing work unnecessarily. The practice of 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.

OMOS currently comprises approximately 25K lines of code. It supports Mach IPC, Sun RPC, and UNIX System V IPC, and manages Mach and HP/UX shared libraries.

It has been mentioned that the ETYMA framework models strong typechecking of module combination. How is typechecking incorporated in the linker? For C dot-o's, we have devised a way in which to extract interface information from dot-o files that have been compiled with debugging information, and typechecking combination at link-time. We have built in the type system of C as a completion of the type framework of ETYMA. With the motivation of generating coercion stubs for compatible encapsulated data, we have implemented a partial order of subtypes of C primitive types based on their coercibility, even though there is no notion of subtyping in the C language. Such type-safe linkage of object files is described in detail in [BLO94]. Another completion of the type framework of ETYMA is described in the following section.

5 An Interface Definition Language

This section describes an interface definition language based on the CORBA IDL (as specified in [COR91]) that we are currently designing as another completion of ETYMA[3].

In the context of distributed systems, an IDL is a descriptive language used to specify the interfaces that client objects call, and service providing objects implement. An IDL compiler generates client "stubs" and server "skeletons" corresponding to legal interface specifications. The stubs provide client implementations the information they need to call service providers. Service providers, in turn, flesh out the implementation outlined in the generated skeletons. At run-time, an underlying object request broker (ORB) manages message traffic between clients and servers, taking care of argument marshaling and unmarshalling.

With CORBA IDL, one can specify interfaces comprising data attributes (constant or variable) and operations (functions), as well as type definitions and exceptions. The IDL specifies a set of basic data types (e.g. short, float, char, boolean), constructed types (e.g. struct, union, enum), template types (e.g. sequence, string), and arrays that can be used to specify members of an interface. An interface can inherit from another with the inheritance operator ":", in which case all members of the inherited interface become members of the inheriting interface, provided there are no conflicts.

[3]The IDL is only partially implemented as yet. We expect that it will be complete by the time the final version of this paper is required. However, the design aspects of the IDL are reasonably complete.

The purpose of the IDL is to specify, adapt, and combine interfaces, so as to generate stubs and skeletons for implementations of those interfaces. Hence, flexibility is a highly desirable property of an IDL. We believe that it is beneficial to extend the CORBA IDL based upon the ETYMA model of interfaces. In fact, the CORBA IDL has several shortcomings that can be ameliorated if it is extended to provide inheritance operators supported by ETYMA. Specifically, we address the following shortcomings:

1. Currently, only type definitions, constants, and exceptions can be redefined in derived interfaces. Function and variable attributes cannot be redefined. Introducing subtyping into the IDL will permit more flexible specification of interfaces. Furthermore, it is important to separate the notions of inheritance and subtyping of interfaces. Therefore, we introduce structural typing into a language which currently supports only identity-based typing.

2. Currently, name conflicts in the case of multiple inheritance is illegal. Support for attribute renaming can solve this problem.

3. Just as it is desirable to build up larger interfaces from smaller ones, it is equally desirable to break up larger interfaces into smaller ones. We introduce an operator project (again, analogous to the notion in relational calculus) on interfaces to support this.

It does not seem necessary to distinguish between defined and declared attributes in a language that deals with pure interfaces. The inheritance mechanism supported by CORBA IDL corresponds to our merge operator on interfaces. As outlined above, we will extend the IDL to support two new operators rename and project. The other operators in ETYMA deal with the *defined bindings* of attributes, so are not relevant in the context of pure interfaces. We will also introduce structural subtyping.

With the extended IDL, we can specify interfaces as shown in Figure 7. In the figure, interfaces B and C singly inherit from A, redefining the op1 operation's return value in each case. Interface D multiply inherits from B and C, resolving a naming conflict with the rename operator. Interface E inherits from an interface derived from (subset of) D by the project operator.

This language design can be incorporated into a completion of ETYMA in a pretty straightforward manner. Briefly, the class design for the completion are as follows. Define class IDLInterface as a subclass of class StdInterface, and define methods merge(IDLInterface), rename(Label), and project(Labels) to return new interface objects after performing the appropriate operations. Implement IDLInterface so that its attributes are only IDL type objects (described below). The type equality and subtyping methods of StdInterface can be reused directly in the IDLInterface class. Furthermore, define methods for the generation of stubs and skeletons from IDL type objects.

```
interface A {
  A op1 ();
  long op2(in long arg1);
};
interface B = A merge {
  B op1 ();
  short op3(in B arg1);
};
```

```
interface C = A merge {
  C op1 ();
  long op4(inout long arg1);
};
interface D =
  (B rename op1 b_op1) merge C;
interface E = (D project op1 op4) merge {
  void op5();
};
```

Figure 7: Example specifications in extended IDL

Create a subclass IDLFunctionType of FunctionType to model function-valued attributes (operations) of interfaces. Subtyping is by contravariance for operations. To keep the language design simple, we can define subtyping to be type equality for all the other types. Subclass NamedType to IDLPrimType to model the primitive types of the IDL, as mentioned for C in Section 3.3. For struct types, which have type identity in IDL, subclass IDLStructType from RecordType and NamedType, and redefine eq for identity-based equality, and le to be eq. Similarly for unions, and the other constructed types. With this set of IDL classes, we can write a parser that instantiates these classes and manipulates objects as driven by source program.

In conclusion, we note that operations on IDL interfaces as defined above can hardly be described as inheritance, since there is no notion of self-reference in the interfaces. We are currently working on incorporating notions such as *SelfType* [Bru92] that introduce recursion into interfaces. The ability to refer to the type of an interface in the specification of its own attributes using *SelfType* is similar to a class' ability to refer to self. We leave this as future work.

6 Related Work

Several O-O frameworks have been developed, initially for user interfaces, and subsequently for many other domains as well [Deu89, VL89, WGM88, CIJ+91]. ETYMAhas a close relationship to compiler frameworks, which comprise classes usually for generating an internal representation of programs. Compiler frameworks fall into two categories: those that represent programs syntactically such as [WBG94], and those that represent programs semantically, such as ours. Compiler frameworks are designed with various objectives, such as for representing abstract syntax, constructing tools for programming environments [BCC+94], or for structuring the compiler itself, e.g. with objects representing phases of the compiler [GR83], or for enabling compile-time reflection via meta-object protocols [KLM94]. ETYMA, while supporting many of the above, is unique in that it is intended to be a reusable architecture for constructing a variety of modular systems.

The Smalltalk-80 system [GR83] is built upon a set of compiler classes that support representing the abstract syntax of programs, as well as an impressive collection of highly intertwined meta-classes that represent the semantics. However, its dynamic meta-circular architecture is tightly coupled with the environment, making it difficult to disentangle the architecture for separate reuse.

The Common Lisp Object System Meta-Object Protocol (CLOS MOP) [KdRB91] supports user-redefinable protocols for meta-objects such as class, instance, generic function, method, etc. CLOS MOP provides the basis for the development of a "space of languages with the default language being a distinguished point in the space." So, in a sense, its architecture is reusable.

However, there are important differences between our approach and previous ones. Our notion of modules is motivated by a desire to uniformly treat the semantics of inheritance. In addition, *encapsulation* is an important semantic requirement in ETYMA, since we believe that it is crucial for software development in the large. Static typing is another important consideration in ETYMA. Furthermore, the ETYMA class interfaces are derived from a rigorous semantic foundation, rather than, for example, from the requirements of diverse language designs already in existence. ETYMA is specifically designed to facilitate the construction of modular systems, but can be used for many purposes that the CLOS MOP has been put to use, notably persistent objects [Lee92].

7 Conclusions and Future Work

We have characterized object-oriented programming as the adaptation and combination of a simple notion of classes, called *modules*. A meta-level architecture for modular systems, realized as a reusable object-oriented framework called ETYMA has been described. ETYMA models classes corresponding to much of the value and type domains of modular languages. Like traditional denotational semantics, which uses functional programming to describe language functionality, ETYMA uses modular programming to describe language modularity.

Central to ETYMA is an austere notion of software modules coupled with a powerful set of inheritance operators to adapt and combine them. As a result, the meta-architecture is reusable in a wide variety of contexts. We have demonstrated its reuse potential by illustrating two significant applications of it: (i) a programmable linker/loader that supports link-time inheritance operations on languages that may not even be O-O, and (ii) an extension of the CORBA interface definition language that supports more flexible adaptation of interfaces. In our experience, not only has the architecture of the framework enhanced the flexibility of its completions, but the completions themselves have contributed to the evolution of the framework.

We see many avenues for future research. We plan to first complete our extended IDL implementation, as well as implement extended IPC functionality with the select operations. There is also potential for incorporating reusable abstractions from the domain of process address spaces, mapping, and shared memory into ETYMA. We also plan to augment the type system in the framework to encompass issues related to the type of self. Futhermore, we are exploring interoperability of the tools built as completions of the framework. For example, the IDL can be used to specify the interfaces and interconnection of object modules, and to automatically generate module composition scripts to be used by OMOS.

Acknowledgements.

We are indebted to Gilad Bracha for his fundamental work in conceiving ETYMA, and his detailed comments on various drafts of this paper. The insights and support of Jeff Law, Robert Mecklenburg, Jay Lepreau, Bryan Ford, Charles Clark and all other Mach Shared Objects project participants are also gratefully acknowledged.

References

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proc. OOPSLA Conference*, Ottawa, October 1990. ACM.

[BCC+94] J. Barton, P. Charles, Y. Chee, M. Karasick, D. Lieber, and L. Nackman. Codestore: Infrastructure for C++ - knowledgeable tools. Presented at the *O-O Compilation* Workshop at OOPSLA, October 1994.

[BL92] 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.

[BL94] Guruduth Banavar and Gary Lindstrom. The design of object-oriented meta-architectures for programming languages. In *Proc. Third Golden West International Conference on Intelligent Systems*, Las Vegas, NV, June 1994. Also available as Technical Report UUCS-94-033.

[BLO94] 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.

[Bra92] Gilad Bracha. *The Programming Language* Jigsaw*: Mixins, Modularity and Multiple Inheritance.* PhD thesis, University of Utah, March 1992. Technical report UUCS-92-007; 143 pp.; ONR 94:1 report.

[Bru92] Kim B. Bruce. A paradigmatic object-oriented programming language: Design static typing and semantics. Technical Report CS-92-01, Williams College, January 31, 1992.

[Cha89] Craig Chambers. Customization: Optimizing compiler technology for self, a dynamically typed object-oriented programming language. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, Jun 21 - 23, 1989.

[Cha92] Steve Chamberlain. libbfd. Free Software Foundation, Inc. Contributed by Cygnus Support, March 1992.

[CIJ+91] Roy H. Campbell, Nayeem Islam, Ralph Johnson, Panos Kougiouris, and Peter Madany. *Choices*, frameworks and refinement. In *Object Orientation in Operating Systems*, pages 9–15, Palo Alto, CA, October 1991. IEEE Computer Society.

[CM89] Luca Cardelli and John C. Mitchell. Operations on records. Technical Report 48, Digital Equipment Corporation Systems Research Center, August 1989.

[COR91] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, December 1991. Revision 1.1.

[CP89] 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.

[Deu89] L. Peter Deutsch. Design reuse and frameworks in the Smalltalk-80 programming system. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability*, volume 2, pages 55–71. ACM Press, 1989.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley Publishing Company, 1995.

[Gin89] Robert A. Gingell. Shared libraries. *Unix Review*, 7(8):56–66, August 1989.

[Gor79] Michael J. C. Gordon. *The Denotational Description of Programming Languages*. Springer-Verlag, 1979.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[HP91] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Proc. of the ACM Symp. on Principles of Programming Languages*, pages 131–142, January 1991.

[JR91] Ralph E. Johnson and Vincent F. Russo. Reusing object-oriented designs. Technical Report UIUCDCS 91-1696, University of Illinois at Urbana-Champagne, May 1991.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, MA, 1991.

[KLM94] Gregor Kiczales, John Lamping, and Anurag Mendhekar. What a metaobject protocol based compiler can do for lisp. Unpublished report. A modified version to be presented at the OOPSLA '94 workshop on O-O Compilation., 1994.

[Lee92] Arthur H. Lee. *The Persistent Object System MetaStore: Persistence Via Metaprogramming*. PhD thesis, University of Utah, June 1992. Technical report UUCS-92-027; 171 pp.

[OBLM93] 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.

[OM92] 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.

[OMHL93] Douglas B. Orr, Robert W. Mecklenburg, Peter J. Hoogenboom, and Jay Lepreau. Dynamic program monitoring and transformation using the OMOS object server. In *Proc. of the 26th Hawaii International Conference on System Sciences*, pages 232–241, January 1993. Also available as technical report UUCS-92-034.

[OMHL94] 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.

[See90] Donn Seeley. Shared libraries as objects. In *Proc. USENIX Summer Conference*, Anaheim, CA, June 1990.

[VL89] John M. Vlissides and Mark A. Linton. Unidraw: a framework for building domain-specific graphical editors. In *Proceedings of the ACM User Interface Software and Technologies '89 Conference*, pages 81–94, November 1989.

[WBG94] Beata Winnicka, Francois Bodin, and Dennis Gannon. C++ objects for representing and manipulating program trees in the Sage++ system. Presented at the *O-O Compilation* Workshop at OOPSLA, October 1994.

[WGM88] A. Weinand, E. Gamma, and R. Marty. ET++: an object-oriented application framework in C++. In *Proceedings of OOPSLA '88*, pages 46–57. ACM, November 1988.

Last revised December 2, 1994