An Application Framework for Module Composition Tools

Guruduth Banavar*
IBM TJ Watson Research Center
30 Saw Mill River Road
Hawthorne, NY 10532 USA
banavar@watson.ibm.com

Gary Lindstrom
Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA
lindstrom@cs.utah.edu

Abstract

This paper shows that class inheritance viewed as composition of recursive namespaces is a broadly applicable concept. We show that several kinds of software artifacts can be modeled as recursive namespaces, and software tools based on a model of composition of namespaces can effectively manage these artifacts. We describe four such tools: an interpreter for compositionally modular Scheme, a compositional linker for object files, a compositional interface definition language, and a compositional document processing tool. We show that these tools benefit significantly from incorporating inheritance-based reuse.

The implementation of tools such as the above share much in common since they are based on the same underlying model. We describe a reusable OO framework for efficiently constructing such tools. Three of the above tools were built by directly reusing the application framework, and the fourth evolved in parallel with it. We provide reuse statistics and experiences with the development of our framework and its completions.

Keywords: inheritance, modularity, module manipulation tools.

*Primary contact. Phone: +1-914-784-7755, fax: +1-914-784-7455.

1 Introduction

Inheritance of classes in object-oriented programming has been touted for enabling significant levels of implementation reuse. Inheritance is widely acknowledged to support reuse via incremental programming — one needs to only program how new classes differ from already existing ones.

One characterization of class-based inheritance is that it is the combination of self-referential namespaces [11]. By carefully designing operations to manipulate such namespaces, a wide spectrum of effects of single and multiple inheritance can be obtained. *Compositional modularity* [3, 6] is such an inheritance model, in which self-referential namespaces, known as modules, can be adapted and composed in various ways to achieve implementation reuse. Compositional modularity supports a stronger and more flexible reuse model than traditional class-based inheritance.

When class-based inheritance is distilled down to a notion of *operations on recursive namespaces*, it becomes possible to explore the breadth of applicability of the concept of inheritance. There is indeed a wide range of software artifacts that can be modeled as recursive namespaces. For instance, it is well known that recursive interface types can be viewed as self-referential namespaces [8, 1]. A traditional compiled object file can also be viewed as a self-referential namespace. Furthermore, structured document fragments can be modeled as self-referential namespaces. Even other artifacts, such as GUI components and file system directories can be regarded as recursive namespaces.

There currently exists a range of tools that manage the range of artifacts mentioned above. However, many such tools are usually based on disparate, and often impoverished, underlying models. In this paper, we argue that it is advantageous to manage the above artifacts from the viewpoint of a well understood model such as compositional modularity, and design tools based on this viewpoint. The primary advantage of such an approach is that the underlying model of such tools can be significantly enriched, and reuse mechanisms akin to inheritance can be supported on the artifacts they manage. Moreover, the uniformity of the underlying model of such tools can be exploited to support better interactions between them.

The model of compositional modularity can be easily and effectively applied within tools that manipulate artifacts such as the ones given above. To demonstrate, we describe four such tools in this paper: (i) an interpreter for a Scheme module system extended to be compositional, (ii) a linker that manipulates compiled object files as compositional modules, (iii) a compiler front-end for an interface definition language with compositional interfaces, and (iv) a document processing system that manipulates documents as compositional modules. We also discuss other tools that could be based on compositional modularity. We show that tools such as the above derive important benefits from incorporating compositional modularity.

Naturally, the implementations of these tools share much in common, since they are all based on compositional modularity. It is therefore beneficial to abstract their common aspects, and realize them as a reusable software architecture. We have designed an OO application framework known as ETYMA that encompasses the reusable architecture of tools based on compositional modularity. The primary utility of the ETYMA framework is that it enables one to easily and rapidly build module composition engines for tools that manipulate a variety of compositional modules. ETYMA consists of more than 40 reusable C++ classes. In this paper, we document the architecture of ETYMA using design patterns and describe the construction of three of the above four tools as direct completions of the framework. We report that significant design and code reuse (between 73 and 91%) was obtained in the construction of the above prototypes as completions of the framework. We also outline our experience with the iterative development of the framework.

The following section provides some background on compositional modularity. In Section 3, we describe the four compositional tools mentioned above. We then describe the design of the ETYMA framework and its completions in Section 4.

2 Background and Related Work

Based on the notion of records and operations on them [9], Cook and Palsberg [11] modeled a class as a self-referential record generating function, e.g., $g = \lambda s.\ \{a_1 = v_1, a_2 = v_2, \ldots, a_n = v_n\}$, also known as a *generator*. The parameter s corresponds to the generator's notion of "self." References to names within the generator are via the s parameter, e.g., $s.a_1$. The fixpoint $Y(g)$ of such a generator, a record, corresponds to an *instance* of the class. Taking the fixpoint of the generator binds the generator's self-references $s.a_x$. The notion of inheritance is modeled as combination of generators, via operators such as merge and override. For instance, the notion of method overriding in generators is defined in terms of record overriding ($\leftarrow_r$ [9]):

$$\mathsf{override} = \lambda g_1.\ \lambda g_2.\ \lambda s.\ g_1(s) \leftarrow_r g_2(s)$$

The crucial aspect of inheritance is that of self-reference manipulation; while combining classes during inheritance, a superclass' notion of self must be properly modified to include that of the subclass.

Based on Cook's work, Bracha and Lindstrom [6] developed a uniform and comprehensive suite of linguistic operations on *modules*, modeled as generators, that individually achieved effects of rebinding, sharing, encapsulation, and static binding. In addition to making previously existing operators explicit linguistic constructs, they define three new operators: hide, freeze, and copy-as. As an example, an attribute of a generator can be copied under another name in order to achieve access to overridden methods, as follows ($\|_r$ stands for record merging [9]):

$$\textsf{copy-as}\ a\ b = \lambda g.\ \lambda s.\ \mathit{let}\ \mathit{super} = g(s)\ \mathit{in}\ \mathit{super}\ \|_r\ \{b = \mathit{super}._r a\}$$

We further augment the above model in [3] to include a notion of hierarchical nesting as a composition operation, feeling that module nestability and separate development must co-exist in a modularity framework without compromising each other. This requires abstracting the environment of a generator, resulting in what we call a *closed* generator, e.g., $g_c = \lambda e.\ \lambda s.\ \{a_1 = v_1, a_2 = v_2, \ldots, a_n = v_n\}$, in which environmental references are made via a separate e parameter. With this, separately developed modules can be *retroactively* nested into conforming modules via a composition operator named nest, defined as follows:

$$\textsf{nest}\ n = \lambda g_{c_{in}}.\ \lambda g_{c_{out}}.\ \lambda e.\ \lambda s.\ \{n = \lambda d.\ g_{c_{in}}(e \leftarrow_r s)\}\ \|_r\ g_{c_{out}}(e)(s)$$

Compositional Modularity. The above concept of closed generators, along with eight primary operations on them, merge, override, rename, copy-as, restrict, freeze, hide, and nest, within an imperative store-based framework with appropriate static typing rules, comprise the model of compositional modularity (or CM for short) [3]. The term composition is used here to mean implementation composition to achieve reuse akin to inheritance. The goal of CM is to get maximal reuse out of small, composable components. The composition constructs given above provide a powerful framework for building larger modules from smaller ones. These constructs can be used in combination to emulate various composite inheritance idioms in existing OO languages. As a result, CM supports a stronger (by virtue of compositional nesting) as well as a more flexible (by virtue of "unbundled," composable operators) notion of reuse than traditional inheritance models.

3 Systems based on Compositional Modularity

To provide a better understanding of how one can apply CM within various tools, we describe four systems based on CM in this section. As mentioned earlier, CM can be layered on top of systems that have a notion of self-referential namespaces and some benefit to be derived from composing them. For systems that have these characteristics, a software tool that manipulates such namespaces using operations of CM can be constructed. However, it must be pointed out that not all eight of the CM operations may be useful or even possible within every system. Nevertheless, we will show in the following sections that enriching a system by incorporating CM gives rise to specific benefits relating to the system's expressive power, flexibility, and/or scope.

3.1 CMS

The first obvious choice for applying compositional modularity is within a modular programming language. In this section, we describe a module system based on CM for the programming language Scheme. Detailed examples using this module system will be provided to help the reader understand CM better.

A module is generally understood to be an independent naming scope. A Scheme module may be regarded as a set of identifiers bound either to locations (variables) or to any of the various Scheme values, including procedures. Procedures may contain references to other name bindings within the module, or to unbound names within the module. (Unbound names correspond to the notion of imported names in traditional module systems. The actual importation can be effected via module combination, described below.) In this manner, a Scheme module may be modeled as a self-referential namespace.

Several module systems for Scheme have been proposed previously [12, 23, 21], but these systems mainly provide a facility for structuring programs via decomposition. However, the ability to *recompose* first-class modules can additionally support design and implementation reuse akin to inheritance in OO programming. Furthermore, the notion of first-class modules and their operations in CM is consistent with the uniform use of first-class values and the expression-oriented nature of Scheme. Consequently, we argue that the incorporation of CM into a module system for Scheme can be very beneficial. In the rest of this section, we describe such a system, named Compositionally Modular Scheme, or *CMS* for short.

Module definition and encapsulation. A module in *CMS* is a Scheme value that is created with the `mk-module` primitive. It consists of a set of attributes (symbol-binding pairs) with no order significance. Attributes that are bound to procedures are referred to as *methods*, borrowing from OO programming. Modules may be manipulated, but their attributes cannot be accessed or evaluated until they are instantiated via the `mk-instance` primitive. The attributes of a module instance can be accessed via the `attr-ref` primitive, and assigned to via the `attr-set!` primitive. A method can access other attributes within its own instance via analogous primitives: `self-ref` and `self-set!`.

(a)

```
(define fueled-vehicle (mk-module
                         ((fuel 0)
                          (empty? (lambda () (= (self-ref fuel) 0)))
                          (fill (lambda () (self-set! fuel (self-ref capacity)))))))
```

(b)

```
(define encap-fueled-vehicle (hide fueled-vehicle '(fuel)))
(describe encap-fueled-vehicle)
    ⟹
    ((empty? (lambda () (= (self-ref <priv-attr>) 0))) (fill ...))
```

(c)

```
(define capacity-module
   (mk-module ((capacity 10)
               (greater-capacity? (lambda (in)
                                    (> (self-ref capacity) (attr-ref in capacity)))))))
(define vehicle (merge encap-fueled-vehicle capacity-module))
```

(d)

```
(define new-capacity
    (mk-module ((capacity 25))))
(define new-vehicle (override vehicle new-capacity))
```

(e)

```
(define v1 (mk-instance vehicle))
```

Figure 1: Basic module operations. (a) Definition via mk-module (b) Encapsulation via hide (c) Combination via merge (d) Rebinding via override, (e) Instantiation via mk-instance.

Figure 1 (a) shows a simple module with three attributes bound to a Scheme variable fueled-vehicle. Note that the fill method refers to an attribute capacity that is not defined within the module, but is expected to be the fuel capacity of the vehicle in gallons.

The primitive hide encapsulates its argument attributes. In Figure 1 (b), the hide expression returns a new module with an encapsulated fuel attribute that has an internal, inaccessible name, shown by the describe primitive as <priv-attr>.

Module combination and instantiation. The module capacity-module given in Figure 1 (c) exports two symbols, including one named capacity. Thus, the module encap-fueled-vehicle can be combined with capacity-module to satisfy the former's "import" requirement, via the primitive merge. The new merged module vehicle in 1 (c) contains four public attributes: empty?, fill, capacity, and greater-capacity?.

The primitive merge does not permit combining modules with conflicting defined attributes, i.e., attributes that are defined to have the same name. In the presence of conflicting attributes, one can use override, which creates a new module by choosing the right operand's binding over the left operand's in the resulting module. For example, the module new-capacity in Figure 1 (d)

(a)	(describe (restrict vehicle '(capacity))) $\Longrightarrow$ ((fill ...) (empty? ...) (greater-capacity? ...))
(b)	(describe (rename vehicle '(capacity) '(fuel-capacity))) $\Longrightarrow$ ((fuel-capacity 10)(fill ... (self-ref fuel-capacity))...)
(c)	(describe (copy-as vehicle '(capacity) '(default-capacity))) $\Longrightarrow$ ((capacity 10)(default-capacity 10)(fill ...(self-ref capacity))...)
(d)	(describe (freeze vehicle '(capacity))) $\Longrightarrow$ ((capacity 10)(fill ...(self-ref <priv-attr>)) ...)

Figure 2: Adaptation. (a) Removing an attribute via restrict (b) Renaming an attribute and self-references to it via rename (c) Copying an attribute via copy-as, and (d) Statically binding self-references to an attribute via freeze.

cannot be merged with vehicle since the two modules have a conflicting attribute capacity. However, new-capacity can override vehicle, as shown.

An instance of the vehicle module, such as v1 in Figure 1 (e), represents exactly the kind of module interconnectivity that can be specified by the use of import/export operations in traditional module systems.

Module adaptation. Besides hide, there are four other primitives which can be used to create new modules by adapting some aspect of the attributes of existing modules. The primitive restrict simply removes the definitions of the given (defined) attributes from the module, i.e., makes them undefined (see Figure 2 (a)). The primitive rename changes the names of the definitions of, *and* self-references to, attributes in its second argument list to the corresponding ones in the third argument. Undefined attributes, i.e., attributes that are not defined but are self-referenced, can also be renamed. An example is shown in Figure 2 (b).

The primitive copy-as copies the definitions of attributes in its second argument list (which must all be defined) with corresponding names in its third argument. An example is shown in Figure 2 (c). The primitive freeze statically binds self-references to the given attributes, provided they are defined in the module. Freezing the attribute capacity in the module vehicle causes self-references to capacity to be statically bound, but the attribute capacity itself is available in the public interface for further manipulation, e.g., rebinding by combination. As shown in Figure 2

(a)	(define vehicle-category (mk-module () ((capacity 10) (type1 (mk-module (...) ((fill (lambda... (env-ref capacity)...))))) (type2 (mk-module (...) ((fill (lambda... (env-ref capacity)...)))))))) (define mycategory (mk-instance vehicle-category)) (define v1 (mk-instance (attr-ref mycategory type1)))
(b)	(define veh-type (mk-module (...) ((fill (lambda ... (env-ref capacity) ...))))) (define new-vehicle-category (nest type3 veh-type vehicle-category))

Figure 3: Nested Modules. (a) Lexical nesting, (b) Compositional nesting via the nest operator.

(d), frozen self-references to capacity are transformed to refer to a private version of the attribute.

Module nesting. In *CMS*, modules may be nested within other modules by binding them to attributes, as shown in Figure 3 (a). Nested modules may refer to name bindings in their surrounding module via the env-ref primitive. Additionally, a seperately developed module may be retroactively nested within another module via the operator nest. An example is shown in Figure 3 (b). The nest expression in the example produces a module that contains the attribute type3 bound to the nested module veh-type just as if it was directly lexically nested.

Composite Inheritance. With the above suite of primitives, several composite inheritance idioms including super-based and prefix-based single inheritance, as well as mixin-based and general forms of multiple inheritance with various types of conflict resolution and sharing strategies can be emulated; please see [3] for a detailed description. To give some insight, Figure 4 pictorially shows how super-based and prefix-based single inheritance can be emulated using CM primitives. Figure 4 (a) shows a "superclass" SUPER with a method METH and self-references to it. An increment DELTA has a redefinition of METH in terms of the previous definition, referred to as METH', as well as some self-references to METH. SUPER and DELTA can be combined to form the "subclass" SUB by using the sequence of operators (copy-override-hide) shown in the figure caption. Similarly, the BETA-style [17] prefixes SUPER and DELTA in Figure 4(b) can be combined using a similar sequence of operations. The difference is that (an adapted version of) the superclass overrides the increment in the case of prefix-based inheritance, as opposed to the reverse for super-based inheritance. Indeed, that is the difference between the two forms of single inheritance.

Two idiomatic sequences of operations in CM have proven to be very useful: copy-override-hide, and rename-merge-hide. These and other idioms of CM will be shown as we proceed.

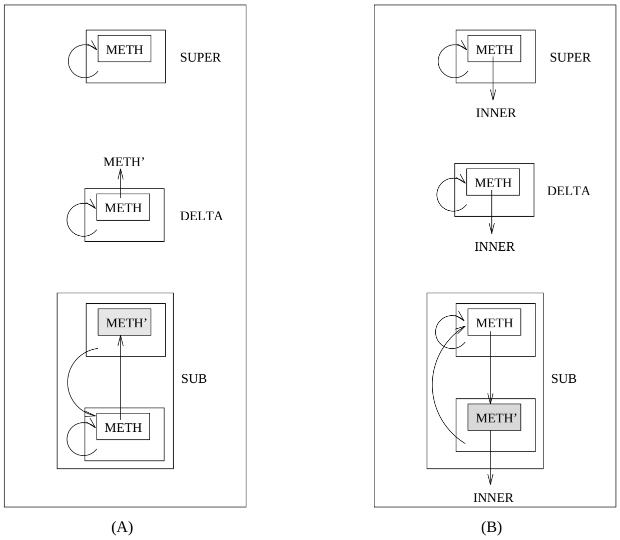

Figure 4: Pictorial representation of single inheritance. (a) Super-based: (hide (override (copy-as SUPER METH METH') DELTA) METH'), (b) Prefix-based: (hide (override (copy-as DELTA METH METH') (rename SUPER INNER METH')) METH').

3.2 Compositional Linking

In this section, we describe the second of the four tools based on CM: a programmable linker.

A separately compiled object file essentially consists of a set of symbols, each associated with data or code. This set of symbols is represented as a symbol table within the object module. Furthermore, there are internal self-references to these symbols which are represented as relocation information within the object module. Thus, the physical notion of an object file may be modeled logically as a self-referential namespace.

The traditional notion of linking object files essentially corresponds to the merge operation in CM. However, the full power of CM made available via a programmable linker can significantly enhance the ability to manage and bind object modules. In particular, facilities such as function interposition, management of incremental additions of functionality to compiled libraries, and namespace management can be made more principled and flexible, as shown below. Consequently, there is much to gain from incorporating CM into a programmable linking tool.

A programmable linker. OMOS [20, 5] is a programmable linker that supports CM for C language object files. OMOS is programmed using a Scheme based scripting language similar to *CMS* above, except that the modules manipulated in this language are compiled object files (dot-o files) as opposed to Scheme modules. A dot-o can be converted into a first-class compositional module via a primitive open-module, manipulated using the CM primitives, and instantiated into

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

Figure 5: Syntax of module primitives.

executable programs (bound to particular points in a process' address space) using the primitive fix.

The syntax of OMOS module primitives is given in Figure 5. Most module operations require transformations on the symbol table of the object file. For instance, the restrict operation essentially 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. Similarly, the rename and copy-as operations modify symbol table entries. The primitive nest is not supported by OMOS, since the notion of nesting is not supported by the C language.

Wrapping. To illustrate the use of the above primitives, this section describes how to achieve several variations of a facility generally referred to as "wrapping." Figure 6 shows a C language service providing module LIB with a function f(), and its client module CLIENT that calls f(). (Although OMOS operates on compiled dot-o files, the C source for modules is shown in the figure for illustration purposes.) Three varieties of wrapping can be illustrated with the modules shown in the figure.

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

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

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

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

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

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

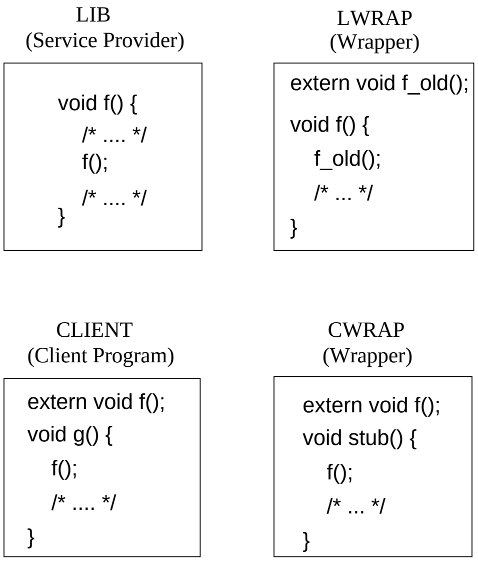

Figure 6: Examples of wrapping in OMOS.

(3) If we want to wrap all calls to f() from CLIENT so that they are mediated via the stub() function of module CWRAP, we can use the following expression:

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

Note that only a particular client module is wrapped, without wrapping the service provider. In the example, renaming the client module's calls to f() produces the desired effect, since the declaration of f() as well as all self-references to it must be renamed.

The idioms given above are in fact the basis of inheritance in OO programming. Scheme macros that perform various kinds of single and multiple inheritance can be used within OMOS just as in *CMS*. In [5], we describe an architecture for OO application development via programmed linkage using OMOS. Specifically, we show how to manage extensions to libraries, how to generate static constructors and destructors, and how to manage the problem of flat namespaces with dot-o files generated from the C langauge.

3.3 Interface Composition

In this section, we describe the third of the four tools based on CM: a compositional interface definition language.

An interface is essentially a naming scope, with labels bound to types. In the case of recursive types, type constituents of the interface can recursively refer back to the interface itself [8, 9]. Thus, an interface can be modeled as a recursive namespace.

Explicit specification and composition of interfaces, as embodied in interface definition languages

(IDLs), is becoming widespread in modern distributed systems [19, 2, 16]. It is useful to specify an interface by *reusing*, i.e., inheriting from, existing interfaces. Reuse facilitates the evolution of interfaces [15] by ensuring that inheriting interfaces evolve in step with the inherited interfaces. It also simplifies maintenance by reducing redundant code. Most importantly, an IDL should be able to express the types of components generated via implementation inheritance in module implementation languages. In fact, it has been shown that inheritance of interfaces generates exactly those types, known as *inherited types*, that correspond to the types of inherited objects [10]. These reasons point to a need for flexible interface inheritance (or composition) mechanisms in IDLs.

A compositional IDL. We have developed a compositional IDL (CIDL) to demonstrate the concepts of compositionality of interfaces. The base type domain of the language consists of primitive types, function types, and record types. Interfaces in this language follow a structural type discipline. Furthermore, interfaces can be recursive, in that a component type of the interface can use the keyword selftype to refer to its own interface. For example, consider a Point module that contains attributes corresponding to rectangular coordinates x and y, a method move for changing the position of the point, and an equality predicate equal. Its interface may be expressed as follows, where recursion is expressed using the selftype keyword. Interface type constituents may refer to sibling type constituents, similar to the notion of sibling member access via *self* in objects. (In this manner, the type of the move operation below can track changes to the x and y constituents.) As a convenience, selftype.x may be abbreviated to x.

```
interface FloatPointType {
  float x, y;
  selftype move (x, y);
  boolean equal (selftype);
}
```

Inheritance is an operation on recursive structures, thus it can be applied to interfaces as well. For instance, the interface FloatPointType above can be extended to have a color attribute using the merge operation, as follows:

```
interface ColorType {
  color_type color;
};
interface ColorPointType = FloatPointType merge ColorType;
```

Although it inherits from FloatPointType, the ColorPointType interface is not a subtype of FloatPointType, due to the contravariance of the equal method. However, ColorPointType shares the same structure as FloatPointType, hence it is known as an *inherited type* of FloatPointType [8, 7].

An important point to note here is that the merge operation on interfaces generates types that correspond to the types of inherited module implementations generated via both the merge *and* override operations. An override operation is defined on interfaces as well, by which type constituents of interfaces may be arbitrarily rebound. The primary motivation for including such an operator is to support a high degree of reuse of existing interface specifications. In the following example, the x and y constituents of FloatPointType are rebound to complex_type; note that this will automatically result in the proper type for the move constituent, due to self-reference.

```
interface ComplexPointType =
  FloatPointType override
    interface {
      complex_type x, y;
    };
```

Type constituents may be rename'd, which results in self-references to get renamed as well. This is useful for resolving name conflicts while performing operations equivalent to multiple inheritance. Furthermore, particular interface constituents may be project'ed. This operation is analogous to the one in relational algebra, and is the dual of the restrict operator presented earlier.

The operator copy-as does not seem very useful in the context of interfaces. Also, the operators freeze and hide do not apply, since interfaces by definition represent the public types of modules. An operation corresponding to nest may be supported, but we are doubtful whether that level of sophistication is required in IDLs.

3.4 Compositional Document Processing

In this section, we describe the fourth of the four tools based on CM: a compositional document processing system.

A structured document may be viewed as a compositional module. Sections within the document correspond to module attributes, with each section comprising a label, associated section heading, and some textual body. Cross references within text to other section labels correspond to self-references. Thus, the document can be regarded as a self-referential namespace.

A large and complex document is often broken down into and composed from smaller pieces. In such scenarios, there are many cases where documents developed for one purpose can be reused for other purposes. For example, a report, such as a user manual, can be composed from several document fragments, such as design documents. A specific scenario of modular document processing that motivated this application of CM was document generation and consumption in the activity of building construction such as that described in [22]. Building architects routinely extract and maintain large bases of document fragments that they reuse, edit, and compose into architectural specifications for delivery to particular clients. As another example, in a document centered industrial process, document fragments are generated at all phases of the process with the objective of producing a number of reports such as inventory statement, parts catalog, assembly reports, process monitoring and quality control documents, etc. Thus, effective document composition tools can be useful in enterprises where several documents are generated, edited, composed, maintained, and delivered in various ways. In such environments, the model of compositional modularity can be used to enhance the composability and reusability of documents.

A tool for composing document modules. We have developed a modular document processing system based on CM named MTEX. MTEX is a programmable facility that helps a document preparer to adapt and compose documents effectively. It is built on top of a restricted version of the LaTeX document preparation system [18]. An MTEX program is a script based on Scheme (as in *CMS*) that describes how LaTeX document modules should be constructed and composed.

An MTEX module is modeled as a generator of an *ordered* set of sections, each of which is a label bound either to a section body, or to a nested module. The section label is a symbolic name that can be referenced from other sections (defined using LaTeX's `\label` command). The section body is a tuple (H, B) where H is text corresponding to the section heading, and B corresponds to the actual text body, which consists of textual segments interspersed with self-references to labels. Given this model of document modules, consider the meaning of the operations of compositional modularity.

The binary operator merge produces a new document module with the sections of its right module operand concatenated to its left module operand, if there are no conflicting labels between the two module operands. Since the order of sections is significant, merge is associative, but not commutative. The binary operator override concatenates two modules in the presence of conflicting section labels. Conflicting sections in the right operand replace corresponding ones in the left operand. Non-conflicting sections in the right operand are appended to the left operand in the same order that they occur in the right operand.

The restrict operator has the usual meaning of removing sections. However, its dual operator project is potentially more useful in the context of document composition. The operators rename and copy-as have the usual meaning. We have chosen not to support encapsulation, i.e., the hide operator, and static binding, i.e., freeze, although it could conceivably have some natural meanings for some applications of document processing. Hierarchical nesting is a very important and useful notion in document structuring. The nest operator supports retroactive nesting of document modules, with one caveat: nested MTEX modules may refer to labels only in surrounding modules, but not to labels within sibling nested modules. Interestingly, the notion of instantiating document modules corresponds to running the LaTeX document processing system on them.

4 The ETYMA Framework and its Completions

Earlier, it was mentioned that tools for systems based on CM can be constructed from a common architecture that encompasses the concepts of CM. In this section, we describe a simple software architecture, an OO framework named ETYMA, that can be effectively *reused* to build tools for a wide variety of systems based on CM such as the ones described in the previous section. Tools constructed from this framework benefit not only from the power and flexibility that the underlying model offers, but also from significant design and code reuse. Thus, ETYMA could significantly reduce the resources spent in developing tools, as well as increase their reliability. Furthermore, ETYMA represents a good model for studying the domain of systems based on CM.

A tool for a system based on CM can be said to consist of a front-end that reads in command and data input, a processing engine that performs CM operations on an internal representation (IR), and an optional back-end that transforms the IR into some external representation. The ETYMA framework is intended for constructing the processing engine along with the IR, rather than for building the front- and back-ends to such systems.

ETYMA is implemented in the C++ language. It is continually evolving, but currently consists of about 45 reusable classes, and approximately 7,000 lines of C++ code. The C++ realization of the ETYMA framework has undergone several iterations over almost two years. In Section 4.3, we outline the major evolutionary stages of the framework.

4.1 Structure of Abstractions

Compositional modularity deals with modules, their instances, the attributes they are composed of, and the types of all the above. Thus, the primary concepts that must be captured by a reusable architecture such as ETYMA are those of *modules*, *instances*, *names*, *values*, *methods*, *variables*, and their corresponding types. However, ETYMA is also a linguistic framework, i.e., a framework from which language processing tools will be designed. Thus, while modeling the above concepts, we must not inadvertently limit their generality. For example, a *method* is a specialization of the general concept of a *function*. Similarly, the concept of a *record* is closely related to that of a

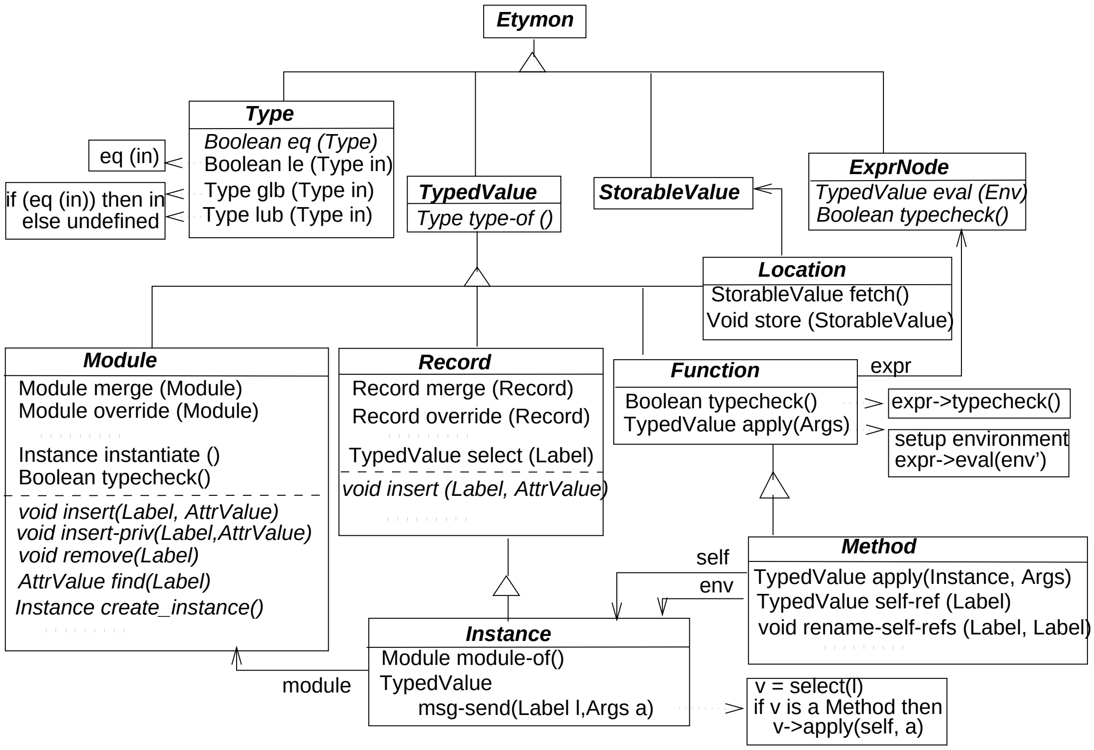

Figure 7: An overview of the abstract classes of ETYMA.

module and an *instance*. We must also be careful in determining the precise relationships between concepts. For example, a module is a record generator whereas an instance is itself a record; thus, the concept of an instance is a subtype of the concept of a record, but both these concepts are not subtype-related to the concept of a module.

The abstractions of ETYMA form two layers. An abstract layer consists of abstract class realizations of the concepts given above. These classes may be used as a "white box" framework (via inheritance) by completions. A concrete layer provides concrete implementations of the abstract classes that can be directly used as a "black box" framework (via instantiation) by completions. This layer, as customary, is meant to increase the reusability of the framework. Only, the important classes in both layers, i.e., those corresponding to modules, instances, and methods, are described in more detail below. (For brevity, we omit classes corresponding to the type system.) We utilize the notion of design patterns [14] to elucidate the structure of the ETYMA framework.

Abstract classes. Figure 7 shows an overview of the abstract classes of ETYMA diagrammed using the OO notation in [14]. Class *Etymon* is the abstract base class of all classes in ETYMA, and classes *TypedValue* and *Type* represent the domains of values and their types respectively.

The abstract class *Module* captures the notion of a compositional module in its broadest conception. Its public methods correspond to the module operators introduced earlier. Within this class, no concrete representation for module attributes is assumed. Instead, the public module operations are implemented as template method patterns in terms of a set of protected abstract methods such as *insert()*, *remove()*, etc. which manage module attributes. Concrete subclasses of *Module* are expected to provide implementations for these abstract protected methods. Two of these are abstract factory method patterns: create_instance, which is expected to return an instance of a concrete subclass of class *Instance* (below), and create_iter, which is expected to return an instance of a concrete subclass of class *AttrIter*, an iterator pattern for module attributes. Thus, the generality of class *Module* results from its use of a combination of the following patterns: template method, abstract factory method, and iterator.

Class *Instance* is a subclass of *Record*; hence it supports record operations, implemented in a manner similar to those of class *Module*. In addition, it models the traditional OO notion of sending a message (dispatch) to an object as *select*'ing a method-valued attribute followed by invoking *apply()* on it. This functionality is encapsulated by a template method pattern msg-send(Label,Args). Furthermore, class *Instance* has access to its generating module via its module member.

The concept of a method is modeled as a specialization of the concept of a function. Class Function supports an apply() method that evaluates the function body. Although class Function is a concrete class, the function body is represented by an abstract class *ExprNode*, a composite pattern. Since a method "belongs to" a class, class Method requires that the first argument to its apply method is an instance of class *Instance*, corresponding to its notion of self.

Concrete classes. Some abstract classes in Figure 7 are subclassed into concrete classes to facilitate immediate reuse. Class StdModule is a concrete subclass of *Module* that represents its attributes as a map. An attribute map (object of class AttrMap) is a collection of individual attributes, each of which maps a name (object of class Label) to a binding (object of class AttrValue). A binding encapsulates an object of class *TypedValue*. This design pattern, a variation of the bridge pattern, makes it possible for completions to reuse much of the implementation of class *Module* by simply implementing classes corresponding to attribute bindings as subclasses of *TypedValue*.

Each of StdModule's attribute management functions is implemented as the corresponding operations on the map. Furthermore, the factory method pattern create_iter of StdModule returns an object of a concrete subclass of class *AttrIter*, class StdAttrIter. Similarly, the factory method pattern create_instance returns an object of the concrete subclass of class *Instance*, class StdInstance.

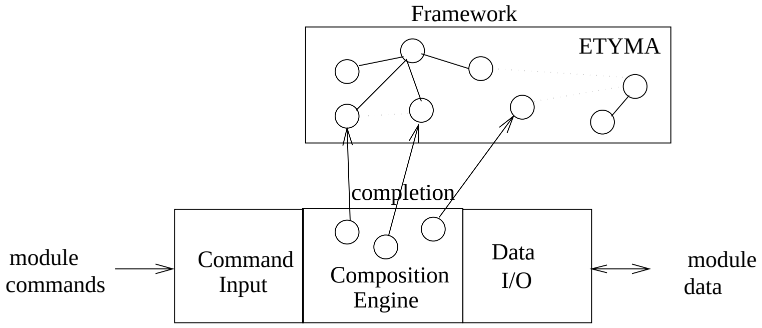

Figure 8: Architecture of completions.

Class StdInstance itself is also implemented using attribute maps.

4.2 Completion Construction

As mentioned earlier, ETYMA can be used to construct the processing engines of tools for compositionally modular systems. In practice, one must first identify the various kinds of name bindings comprising namespaces in the system. One can then identify generalizations of these concepts specified as classes in the ETYMA framework. For each such general ETYMA class, one must then subclass it to implement the more specific concept in the system. Once this is done, concrete classes in the framework corresponding to modules, instances, and interfaces can usually be almost completely reused, due to the bridge pattern mentioned above.

Architecturally, tools constructed as completions of ETYMA have the basic structure given in Figure 8. The command input component reads in module commands that direct the composition engine. The data input component creates the internal representation (IR) of compositional modules by parsing module source data and instantiating appropriate framwork/completion classes. The optional data output component transforms IR into a suitable output format. The composition engine itself is derived from the ETYMA framwork, and comprises classes (data and composition behavior) corresponding to module related entities. In the following subsections, three tools derived from the ETYMA framework in such a manner are described.

4.2.1 An Interpreter for CMS

The *CMS* interpreter consists of two parts: a basic Scheme interpreter written in the C language, and the module system, implemented as a completion of ETYMA. The basic Scheme interpreter itself was derived from a publicly available scriptable windowing toolkit called *STk* [13]. The interpreter implementation exports many of the functions implementing Scheme semantics, thus making it easy to access its internals. Furthermore, the interpreter was originally designed to be

Reuse parameter		New	Reused	% reuse
Module system only	Classes	7	25	78
	Methods	67	275	80.4
	Lines of Code	1550	5000	76.3
Entire interpreter	Lines of Code	1800	20000	91.7

Table 1: Reuse of framework design and code for *CMS* interpreter.

extensible, i.e., new Scheme primitives can be implemented in C/C++ and easily incorporated into the interpreter. Thus, in order to implement *CMS*, Scheme primitives implementing concepts of compositional modularity such as mk-module, mk-instance, self-ref, merge, etc. were implemented in C++ and incorporated into the interpreter.

The class design for the *CMS* module system completion is as follows. Attribute bindings within *CMS* modules can be Scheme values, variables, or methods. These can be modeled as subclasses of framework classes PrimValue (not shown), Location, and Method respectively. The method subclass need not store the method body as a subclass of *ExprNode*; instead, it can simply store the internal representation of the Scheme expression as exported by the interpreter implementation. Additionally, the method subclass must define methods corresponding to the *CMS* primitives self-ref, self-set!, etc. which call similar methods on the stored self object.

With the classes mentioned above, the implementation of class StdModule can be almost completely reused for implementing *CMS* modules. However, methods to handle *CMS* primitives mk-module, mk-instance, etc. must be added. The only modification required is to redefine create_instance to return an object of an appropriate subclass of class StdInstance. This subclass needs to implement code for *CMS* primitives such as attr-ref and attr-set!.

Figure 1 shows several measures of reuse for the *CMS* module system implemented as a completion of ETYMA. The percentages for class and method reuse give an indication of *design* reuse, since classes and their methods represent the functional decomposition and interface design of the framework. On the other hand, the percentages for lines of code give a measure of *code* reuse.

4.2.2 A Compiler Front-end for CIDL

Although we have not described the classes in ETYMA that relate to types, there is a comprehensive set of reusable classes that correspond to the notions of interfaces, record types, function types, etc. All type classes are subclasses of the abstract superclass *Type* in Figure 7 which defines

Reuse parameter	New	Reused	% reuse
Classes	5	22	81.5
Methods	20	155	88.6
Lines of Code	1300	3600	73.5

Table 2: Reuse of framework design and code for IDL front-end.

abstract methods for type equality, subtyping, and for finding bounds of type pairs in a type lattice. Interfaces correspond to the types of modules, and support predicate methods that implement the typechecking rules for each of the module operators. Furthermore, abstract class *Interface* and its concrete subclass StdInterface are implemented in a manner similar to classes *Module* and StdModule.

Briefly, the class design for the CIDL front-end completion are as follows. Attribute bindings within interfaces can be base types, function types, or record types, designed as subclasses of the corresponding generic classes in the framework. Define class IDLInterface as a subclass of class StdInterface, and define methods merge (IDLInterface), rename (Label,Label), etc. to return new interface objects after performing the appropriate operations. Furthermore, the notion of the recursive type selftype is implemented as the special framework class SelfType (a singleton pattern). Recursive type equality and subtyping methods of StdInterface, which implement the algorithms given in [1] can be reused directly in the IDLInterface class. Design and code reuse numbers for this completion prototype are given in Table 2.

4.2.3 An Interpreter for MTEX

The STk-derived interpreter was used for MTEXin a manner similar to *CMS*. The subclasses of ETYMA created to construct the MTEX module engine are: TexLabel of Label, Section of Method, TexModule of StdModule, SecMap of AttrMap, and TexInterface of StdInterface. Also, a new class Segment that represents a segment of text between cross-references was created. Approximate design and code reuse numbers for the MTEX implementation are shown in Table 3.

4.3 Framework Evolution

The very first version of ETYMA was almost fully concrete, and was designed to experiment with a module extension of the C language. It consisted only the notions of modules, instances, primitive values, and locations, along with a few support classes. No front and back ends were constructed.

Reuse parameter		New	Reused	% reuse
Module system only	Classes	6	20	77
	Methods	36	231	86.5
	Lines of Code	1600	4400	73.3
Entire system	Lines of Code	1800	19400	91.5

Table 3: Reuse of framework design and code for building MTEX.

The next incarnation of ETYMA was used to build a typechecking mechanism for C language object modules, described in [4]. This experiment solidified many of the type classes of ETYMA. However, at this point, ETYMA was still primarily a set of concrete classes. The third incarnation was used to *direct* the reengineering of the programmable linker/loader OMOS described earlier. In this iteration, the framework was not directly used in the construction of OMOS (due to practical, not technical constraints), but it evolved in parallel with the actual class hierarchy of OMOS. However, the design of OMOS classes follow that of ETYMA closely. Also, much of the framework, including the abstract and concrete layers, developed during this iteration.

The fourth iteration over ETYMA was the construction of *CMS* completion. There were few changes to the framework classes in this iteration; these were mostly to fix implementation bugs. However, some new methods for retroactive nesting were added. Nonetheless, the *CMS* interpreter was constructed within a very short period of time, and resulted in a high degree of reuse. The next iteration was to design and implement an IDL compiler front-end. There were almost no modifications to the framework; additions included selftype related code. The sixth, and most recent, iteration over ETYMA has been to build the document composition system. There were no changes to the framework.

The first three iterations essentially evolved the framework from a set of concrete classes to a reusable set of abstract and concrete classes, thus crystallizing the reusable functionality of the framework. From the fourth iteration onwards, the framework was mostly reused, with some additions, by very few modifications. As the observed reusability of the framework increased, measurements were taken to record the reuse achieved, as shown in the tables earlier.

5 Conclusions and Future Work

We have shown in this paper that OO inheritance viewed as operations on recursive namespaces is a broadly applicable concept. Specifically, we have shown how to apply compositional modularity (CM), a model that defines a comprehensive suite of operations on recursive namespaces, to a variety of software artifacts such as Scheme modules, compiled object files, interfaces, and document fragments.

We have described four tools that can help effectively manage these software artifacts. These are: (i) an interpreter for the programming language Scheme extended with the notion of compositional modules, (ii) a linker that manipulates compiled object files as compositional modules, (iii) a compiler front-end for a language with compositional interfaces, and (iv) a document processing system that manipulates documents as compositional modules. Furthermore, we show that these systems benefit significantly by incorporating concepts of composition (i.e., inheritance).

The implementation of tools for systems based on CM share a lot in common. Hence, we argue that a reusable software architecture for such tools is beneficial. We describe a reusable OO framework named ETYMA from which tools such as the above can be efficiently constructed. ETYMA currently comprises about 45 reusable C++ classes in 7000 lines that evolved over six iterations. Three of the above tools were built by directly reusing ETYMA, resulting in significant levels (between 73 and 91%) of design and code reuse.

Many other tools can be based on CM and can be built by completing ETYMA. Naturally, CM can be applied within other programming language processors: compiler and interpreters for modular and non-modular languages. There is also an abundance of software artifacts that can be viewed as recursive namespaces, and that have a useful notion of composition. For example, tools that manage GUI components viewed as compositional entities are conceivable. File systems that view directories as recursive namespaces (i.e., filenames bound to file contents that refer to other filenames) could also be useful. We also speculate that the commonality of the underlying models of such tools can be exploited for supporting interoperability among them.

Acknowledgements. We gratefully acknowledge support and several useful comments on this work from Jay Lepreau, Bjorn Freeman-Benson, Gilad Bracha, Bryan Ford, Doug Orr, Robert Mecklenburg, and Nevenka Dimitrova.

References

[1] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4), September 1993.

[2] Joshua Auerbach and James Russell. The Concert signature representation: IDL as intermediate language. In Jeanette Wing, editor, *Proc. of Workshop on Interface Definition Languages*, pages 1 – 12, January 1994. Available as August 1994 issue of ACM SIGPLAN Notices.

[3] Guruduth Banavar. *An Application Framework for Compositional Modularity.* PhD thesis, University of Utah, Salt Lake City, Utah, 1995.

[4] 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.

[5] Guruduth Banavar, Douglas Orr, and Gary Lindstrom. Layered, server-based support for object-oriented application development. In Luis-Felipe Cabrera and Marvin Theimer, editors, *Proceedings of the Fourth International Workshop on Object Orientation in Operating Systems*, pages 2–11, Lund, Sweden, August 14 – 15 1995. IEEE Computer Society. Also available as TR UUCS-95-007.

[6] 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.

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

[8] P. Canning, W. Cook, W. Hill, and W. Olthoff. Interfaces for strongly-typed object-oriented programming. In Norman Meyrowitz, editor, *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 457–467, 1989.

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

[10] William Cook, Walter Hill, and Peter Canning. Inheritance is not subtyping. In Carl Gunter and John Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, pages 497 – 517. MIT Press, 1994.

[11] 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.

[12] Pavel Curtis and James Rauen. A module system for Scheme. In *Conference Record of the ACM Lisp and Functional Programming.* ACM, 1990.

[13] Erick Gallesio. STk reference manual. Version 2.1, 1993/94.

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

[15] Graham Hamilton and Sanjan Radia. Using interface inheritance to address problems in system software evolution. In Jeanette Wing, editor, *Proc. of Workshop on Interface Definition Languages*, pages 119 – 128, January 1994. Available as August 1994 issue of ACM SIGPLAN Notices.

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

[17] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. The BETA programming language. In *Research Directions in Object-Oriented Programming*, pages 7 – 48. MIT Press, 1987.

[18] Leslie Lamport. LaTeX*, a Document Processing System.* Addison Wesley Publishing Company, Reading, MA, 1986.

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

[20] 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.

[21] Jonathan Rees. Another module system for Scheme. Included in the Scheme 48 distribution, 1993.

[22] Wayne Rossberg, Edward Smith, and Angelica Matinkhah. Structured text system. US Patent Number 5,341,469, August 1994.

[23] Sho-Huan Simon Tung. Interactive modular programming in Scheme. In *Proceedings of the ACM Lisp and Functional Programming Conference*, pages pages 86 – 95. ACM, 1992.