Module Management As A System Service[1]

Gilad Bracha[2], Charles F. Clark[3], Gary Lindstrom[3,4] and Douglas B. Orr[3]

July 1993

Abstract

A new approach to program modularity is presented, based on three key ideas: (i) elevating module management to a pervasive system service, (ii) casting the notion of *module* as an abstract data type, and (iii) representing modules as objects for programmed manipulation and retention. The result is an innovative system architecture for program construction, with flexible capabilities for name space management, inter-process sharing, and incremental loading and relinking. The necessary reflective powers are provided by a permanently running system process exporting a wide repertory of module manipulation and delivery services. This approach draws on insights from both programming languages and system architecture. From programming languages, we have taken inspiration from the power of inheritance in object-oriented programming languages, and the importance of types as a basis for secure, evolvable module interfaces. From system architecture, we have profited from the server process approach to system organization, and respect for the crucial role of module binding as a system flexibility governor. A case study applying this architecture to C++ is presented, as well as a brief assessment of its role in a more comprehensive architecture for persistent object management.

1 Module Management: What, How, and By Whom?

Modularity and abstraction go hand-in-hand as the primary principles for building soundly conceived, correctly implemented, effectively maintainable and adaptable software systems. Programming language researchers and designers have pioneered the formulation and articulation of these principles in semantically rich and expressive language structures.

The effectiveness of any approach to software modularity is directly governed by its capability to weave modules into a well-engineered system fabric. Hence the ability to abstract over modules, flexibly combine them to achieve systematic development and reuse goals, and instantiate them under various pragmatic limitations becomes a crucial system versatility concern. In recognition of the fundamental nature of module manipulation as a software engineering activity, we advocate

[1]This research was sponsored by the Advanced Research Projects Agency (DoD), monitored by the Department of the Navy, Office of the Chief of Naval Research, under Grant number N00014-91-J-4046. The 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 Advanced Research Projects Agency or the US Government.

[2]Horizon Technologies of New York, Inc. Email: gilad@cs.utah.edu.

[3]University of Utah. Email: {clark,lindstrom,dbo}@cs.utah.edu.

[4]Primary contact author. University of Utah, Computer Science Department, Salt Lake City, UT 84112 USA. Phone 801-581-5586, fax 801-581-5843.

meeting its needs through a module management layer positioned between basic operating system support and language-specific program development tools. Our approach is strongly motivated by the modularity advances in modern programming languages, but does not rely on an intimate connection with any particular source language.

1.1 What Is Module Management?

In simplest terms, module management involves the definition, combination, specialization, and instantiation of independently comprehensible (compilable) program units. This need, with varying degrees of sophistication, has been acknowledged since the earliest higher level languages (e.g. Fortran), and indeed, before then at the assembly language level. Modularity of some form is now considered essential in any language intended for serious scale use. Many important varieties have been formulated within individual programming languages (e.g. packages in Ada and Common Lisp, modules in Modula-3 and Standard ML), as well as within broader contexts (e.g. classes in object-oriented (O-O) languages, and relocatable object files under Unix).

If one looks beyond these domains, however, the modularity landscape is bleaker. Clearly, the advent of distributed computing and *open system* interoperability has demanded more meticulous care in designing software component interfaces and connection mechanisms. However, the development of individual modules has remained mired in decades-old object file, linker and loader technology. We seek to redress this imbalance by adapting the most important language-based advances in program modularity for delivery in a language-neutral manner within conventional operating systems and programming environments.

1.2 How Should Modules Be Managed?

We advocate a new approach to module management, in keeping with today's emphasis on component-based software engineering. In our view, the *definition* of modules is clearly a programming language issue, but the *manipulation* of modules should be viewed as a systemically available resource management function. We stop short, however, of advocating redesign of existing operating systems or hardware architectures — believing instead that the fruits of this approach may be harvested in a more evolutionary manner.

We are attacking this goal by exploiting three key ideas:

1. *Elevating module management to a pervasive system service,* thereby emphasizing that the concerns of module manipulation transcend individual programs, languages or users.

2. *Casting the notion of module as an abstract data type,* thereby applying abstraction to the module notion itself, and facilitating the application of generic manipulation methods without attenuating the importance of language-specific module features.

3. *Representing modules as objects for programmed manipulation and retention,* thereby cul-

tivating a fertile middle ground between full reflection (programs manipulating themselves) and the rote-dull directives of current *program binders* (linkers and loaders).

1.3 Who Should Manage Modules?

In modern software system architectures, pervasive system resources are managed by servers. In this spirit, we advocate empowering a permanently running system process with module management responsibilities. In comparison to the prevalent *utility function* view of program binding, this approach opens many new possibilities, including:

True programmability: The server can respond in open-ended ways to client direction — indeed, it can be a genuine *interpreter* of module manipulation scripts (programs).

State: As a permanently running process, the server can retain cached information about client service patterns — current and past. In particular, an individual client can dialogue with the server throughout the former's lifetime. This possibility has the salutary effect of treating *static* and *dynamic* service requests with uniformity and continuity.

System-wide purview: As a standard system process, the server can perform module manipulation with an eye toward larger resource optimization objectives. These can extend from incremental optimization (e.g. locality-enhancing module reorganization) to sharing (remapping) of previously delivered modules (e.g. *shared libraries*) [Gin89].

Distributed access: Seamless access of remote server processes is a basic organizational concept in networked multi-computers. Such widespread access to a module management process encourages inter-host module sharing and systematic module adaptation in support of heterogeneity, as well as ease of integration into existing software architectures.

2 Modules As Objects

Given its fundamental importance in the software engineering workplace, we assert that effective module management must be accorded (i) a sound and broadly applicable conception, including formal semantics, and (ii) be supported by flexible and powerful computational tools. Point (i) leads us to view modules as instances of abstract data types, which is the focus of this section. Point (ii) is addressed by our system server idea, which is developed further in §3.

2.1 An Abstract Data Type For Modules

A module is a familiar program structuring notion involving encapsulating functions and data, controlling visibility, presenting a typed interface to clients, admitting instantiation, etc. Recent work [BL92] has formalized the module notion as an abstract data type (ADT), customizable to a broad range of specific modularity notions, ranging from O-O classes to conventional Unix object

```
class module
  function make_module(value_bindings: List[(label, value_or_module)]): module;
  function interface_of(): interface;
  function merge(b: module): module;
  function override(b: module): module;
  function restrict(l:label): module;
  function project(l: List[label]): module;
  function rename(l1, l2:label): module;
  function freeze(l:label): module;
  function freeze_all_except(l: List[label]): module;
  function hide(l:label): module;
  function show(l: List[label]): module;
  function copy_as(l1, l2:label): module;
  function instantiate(): instance;
  function parse_module(s: stream): module;
end;
```

Figure 1: module as an ADT in Jigsaw

files. This is distinct from the familiar notion that modules (e.g. classes) can be used to introduce user-defined abstract data types. Rather, the concept of module *itself* is cast as an abstract data type, in the same sense one might characterize stack as an abstract data type.[1]

A module is a self-referential (recursive) scope of attributes, with some attribute names bound to values (i.e. defined) and the remainder simply declared and typed. Modules thus have no free variables, and we assume that declarations are sufficient for separate compilation of modules. Our module ADT provides a constructor function make_module, through which individual modules (cf. classes) are created. Once created, a module can create instances (cf. objects) via an instantiate function, specified (but perhaps not implemented) by the module ADT. Most importantly, the module ADT defines language-independent combinators applicable to well-formed modules. Metaphorically, these combinators permit the deft reshaping and fitting together of modules, much as a jigsaw tool may be used to craft the pieces of a wooden puzzle. In this spirit, we dub the language of module ADT operators Jigsaw.

Our module ADT (see Fig. 1) is *abstract* in several dimensions:

- It contains *operators* that are declared but not defined. This accommodates reification for language-specific module concepts, while retaining generic manipulability through the ADT interface.
- It uses declared but undefined *auxiliary classes,* which provide customizable realizations of

[1]For our purposes, a rudimentary notion of ADT suffices — we simply rely on name space encapsulation and visibility control. Deeper ADT issues, such as appropriate notions of type abstraction, e.g. existential typing, are raised in §6.3.

interfaces to pertinent source language features, such as the *base language* (non-module) type system.

- It provides for the construction of modules which are *themselves* abstract, i.e. contain attributes which are declared but not defined (imported; cf. extern declarations in C).

Hence our module specification is a *framework* in the sense of [JR91]. Indeed, the possibility of exploiting this property to implement a family of language processors sharing a generic notion of modularity is an important thrust of our research (briefly sketched in §6.2; this motivates the parse_module constructor intended for scanning the surface syntax of modules).

2.2 Module Manipulation Under Jigsaw

The essence of Jigsaw's power lies in its ability to manipulate modules in a manner that *unbundles* and *generalizes* the many roles of inheritance in O-O languages. For example, the class notion in C++ occupies a particular point in a space of module manipulation strategies. The C++ class concept embodies a fixed, intimately interacting set of policy decisions on such issues as (i) module (class) definition, typing and combination; (ii) control of visibility and conflicts on attribute names; (iii) sharing and rebindability of attribute values, and (iv) typing and subtyping. These features are managed in a largely separable manner by Jigsaw's suite of module combinators, a subset of which are informally defined in Fig. 2.

A precise semantics for Jigsaw module manipulation, inspired by the typed record calculus [CM89, HP91], is given in [Bra92]. Jigsaw modules are *structurally typed,* i.e. a module type (interface) is a set of attribute names with associated types. For two module types τ and σ, τ is a subtype of σ ($\tau \leq \sigma$) iff (i) the attribute names in τ are a superset of those of σ, and (ii) for each attribute name $\mathcal{A}$ in σ, the type of $\mathcal{A}$ in τ is a subtype of the type of $\mathcal{A}$ in σ, either by the subtyping notion imported from the base language, or recursively in the case of module types.

2.3 An Example: General Function Interposition

The Jigsaw approach offers the important advantage that O-O program adaptation and reuse techniques (*inheritance*, in all its meanings) can be applied to *legacy code* written in non-O-O languages. We demonstrate this by showing how general function interposition can be achieved in Jigsaw.

Function interposition is a very powerful system configuration device, whereby all references to a function f are rebound to an impostor function. In turn, the new function can make references to the prior definition of f (as well as itself, in the general recursive case). In this fashion, new definitions of functions can be made to replace old ones, or inserted *between* the references to the function and its original definition. Interposition has many uses, including debugging, adaptation of previously compiled code to new data formats, and conversion of object identifiers to valid

Name	Sample Module
$\mathcal{O}_1$	{int x; fun f (int y) = g(g(y)); fun g (int z) = z+x}
$\mathcal{O}_2$	{int x = 13; fun q (real z) = z*z}
$\mathcal{O}_3$	{int y = 15; fun g (int w) = w-y}

Operation	Result
$\mathcal{O}_1$ copy f as h	{int x; fun f (int y) = g(g(y)); fun g (int z) = z+x; fun h (int y) = g(g(y))} *(A definition copy is added)*
$\mathcal{O}_1$ freeze g	{int x; fun f (int y) = g(g(y)); fun g (int z) = z+x } *($\mathcal{O}_1$ is unchanged, but* g *becomes non-rebindable)*
$\mathcal{O}_1$ hide g	{int x; fun f (int y) = g'(g'(y)); fun g' (int z) = z+x} *(Component* g' *is not externally visible)*
$\mathcal{O}_1$ merge $\mathcal{O}_2$	{int x = 13; fun f (int y) = g(g(y)); fun g (int z) = z+x; fun q (real z) = z*z} *(Declarations and definitions collected & matched; conflicts disallowed)*
$\mathcal{O}_1$ override $\mathcal{O}_3$	{int x; fun f (int y) = g(g(y)); int y = 15; fun g (int w) = w-y} *(Merge with conflicts resolved in favor of right operand)*
$\mathcal{O}_1$ rename g to h	{int x; fun f (int y) = h(h(y)); fun h (int z) = z+x} *(Declaration and all uses consistently renamed)*
$\mathcal{O}_1$ restrict g	{int x; fun f (int y) = g(g(y)); fun g : int → int} *(Declaration stripped of its definition)*
$\mathcal{O}_1$ show f	{int x'; fun f (int y) = g'(g'(y)); fun g' (int z) = z+x'} *(Complement of* hide *—* x' *and* g' *are hidden)*

Figure 2: Sample Jigsaw operators.

addresses ("swizzling" [Mos92]). Although function interposition (by various names and devices) is a familiar construct in O-O programming, it is beyond the capability of conventional linkers because it fundamentally relies on name rebinding *within* a module.

The general case of interposition can be achieved through the Jigsaw operations copy-as, override, and hide. To illustrate, suppose we wish all invocations of a function f to be redirected to a new definition of f using (in the general case) both its own definition, which it refers to as f, and the existing definition of f, which it refers to as f_{old}. Given the existing module

$$\mathcal{O} = \{\ \mathsf{g} = \alpha(\mathsf{f},\ \mathsf{g}),\ \mathsf{f} = \beta(\mathsf{f},\ \mathsf{g})\ \}$$

and the interposing function module

$$\mathcal{O}' = \{\ \mathsf{f} = \gamma(\mathsf{f}_{old},\ \mathsf{f})\ \}$$

(for brevity we omit the declaration of f_{old}), the desired interposition result is

$$\mathcal{O}_{result} = \{\ \mathsf{g} = \alpha(\mathsf{f},\ \mathsf{g}),\ \mathsf{f}_{old} = \beta(\mathsf{f},\ \mathsf{g}),\ \mathsf{f} = \gamma(\mathsf{f}_{old},\ \mathsf{f})\ \}$$

where f_{old} is hidden (removed from the interface of $\mathcal{O}_{result}$). $\mathcal{O}_{result}$ can be obtained by the Jigsaw operator sequence shown in Fig. 3. We of course wish to preserve the original interface to f, and, indeed, cascaded interposition works as expected.

1. **(Copy-as)** $\mathcal{O}_1 = \mathcal{O}$ copy f as f_{old}
   Result: $\mathcal{O}_1 = \{$ g $= \alpha$(f, g), f $= \beta$(f, g), $\mathsf{f}_{old} = \beta$(f, g) $\}$

2. **(Override)** $\mathcal{O}_2 = \mathcal{O}_1$ override $\mathcal{O}'$
   Result: $\mathcal{O}_2 = \{$ g $= \alpha$(f, g), f $= \gamma$(f_{old}, f), $\mathsf{f}_{old} = \beta$(f, g) $\}$

3. **(Hide)** $\mathcal{O}_{result} = \mathcal{O}_2$ hide f_{old}
   Result: $\mathcal{O}_2$ with f_{old} removed from its interface, as desired.

Figure 3: Function interposition via Jigsaw operations.

```
(hide "f_old"
   (override
      (copy-as "f" "f_old" /u/dbo/O.o)
      /u/dbo/O_prime.o))
```

Figure 4: An OMOS module blueprint.

3 OMOS, A Module Manipulation Server

Regarding modules as instances of an ADT leads immediately to the viewpoint that modules should be manipulated as objects. We are investing these reflective powers in OMOS, an Object/Meta-Object Server [OM92, OMHL93]. The capabilities of OMOS are now described, considering in order (i) module combination ("linking"), (ii) module instantiation ("loading"), and (iii) module caching ("transparent sharing and reuse").

3.1 Module Combination Under OMOS

As its name suggests, OMOS represents modules in two forms: (i) as *extensional objects*, i.e. relocatable *program fragments*, typically compiler emitted object files, and (ii) as *intensional meta-objects* comprising *blueprint expressions* denoting modules via Jigsaw composition operations.

OMOS blueprints use a Lisp-like syntax, illustrated in Fig. 4 by an OMOS specification of the function interposition example in Fig. 3. The Jigsaw expression in the blueprint of a meta-object may be regarded as prescribing a module's *physical derivation* (linkage structure).

3.2 Module Instantiation Under OMOS

From a system architecture perspective, the programming language notion of *module instantiation* becomes the service of *module delivery* to client processes. As an operational requirement of that role, OMOS must deal with address space management. This is accomplished through execution of module operations in conjunction with a constraint subsystem that controls placement of objects within the client's address space. OMOS module construction requests include address space binding (mapping) directives expressed in a constraint sublanguage. These directives can be *absolute* (anchoring a module to a fixed position), *bounded* (mandating location above or below a given boundary), or *compound* (involving union, intersection, and complementation of other constraints).

Often the delivery of an acceptably bound module can be expedited by (i) binding a previously linked but relocatable version of the desired module; (ii) remapping a previously relocated and retained module (described next), or (iii) a combination of the two, whereby optimizations of (i) and (ii) are applied to cached intermediate results of a blueprint-driven module construction.

3.3 Module Caching Under OMOS

OMOS's ongoing execution and persistent state permit optimization of service requests through transparent caching of previously constructed and bound modules. Hence the distinction between extensional and intensional module representation described in §3.1 is blurred to advantage behind the scenes. This has great importance if modules are to serve as libraries shared across multiple clients [Jor90]. Hence OMOS is a "loader with a memory" — which is very helpful within a client session in support of dynamic linking (considered in §4.1).

Moreover, the constraint system permits concurrent users of a module to physically share the same set of methods, as is done by shared libraries. That is, OMOS service requests typically admit various *degrees of freedom* in client address space mapping, thereby enhancing the possibility that a previously constructed, cached instance of the module will be suitable for reuse. In effect, the flexibility of modern shared library schemes is obtained without incurring the overhead or added complexity of compiler-supported techniques such as position independent code [OBLM93]. This builds on the insights of Seeley [See90], who develops in detail the broad correspondences between O-O language and shared library implementation technologies.

4 Advanced Module Manipulation Services

We view the services described in §3 as basic capabilities to be reasonably expected of any module manipulation process. More advanced services can be postulated beyond those. We briefly examine two here, primarily motivated by our needs in the *Mach Shared Objects* persistent object base project [LK92].

4.1 Dynamic Linking and Loading

Dynamic linking and loading can be viewed as module manipulation where one operand is already loaded and undergoing execution. This extension introduces pragmatic considerations in the use of Jigsaw module operations beyond the static case described in §3. A major concern is whether symbols to be rebound in the executing module have references in *data segments* or *code segments* (*text* in Unix jargon). Due to limitations on execution state modification, including instruction cache clearing, it is impractical to modify symbol bindings in code segments. In addition, there are semantic problems associated with changing the definition of an interface that may hold important program state, or may even be concurrently in use by multiple threads. This restriction is accommodated cleanly by viewing symbols referenced from the executing module's code segments as having been subjected to Jigsaw's freeze operation (see Fig. 2). This renders their bindings *read-only*, but does not hide them, so references in the dynamically loaded module can be bound to them.

In the simplest case, dynamic linking and loading involves a run-time merge operation between loaded module and the module being loaded. More sophisticated effects can be obtained through more powerful Jigsaw operators such as override, and compound operations such as interposition. Frozen symbols can be manipulated through operations that only affect the loading module's symbolic interface (e.g. hide and rename-to). Operations that cause binding or rebinding of symbols, such as restrict, override and merge, may not be applied if the binding of a frozen symbol would be affected.

4.2 Modules As Persistent Objects

The lifetime of modules clearly exceeds that of the programs that construct and import them. By definition, then, modules must be persistent. At a minimum, this can be achieved by their retention in a traditional file system. However, the presence of (i) OMOS as a continuing system process, (ii) our view of modules as objects, and (iii) our larger interest in understanding the ramifications of a distributed object store, raise new possibilities for achieving module persistence.

Let us pause to inventory the many module-related entities that are candidates for retention. Within OMOS we have fragments (compiler emitted object files), OMOS meta-objects, cached modules (constructed from meta-objects), and miscellaneous state information. For applications in a particular O-O language, we may have persistent class instances, class description objects (e.g. C++ *dossiers* [IL90], see §5) and class implementation modules.

Two fundamental design questions arise concerning persistence in this diverse world of objects:

1. To what extent should these varieties of objects be treated *uniformly* or *differentially,* and
2. Where should modules physically reside?

Our position on the first question is that *the most important practical needs are met by recognizing three levels of object reflection:* (i) application program objects (e.g. class instances), (ii) modules as objects (e.g. class implementations), and (iii) objects describing system heterogeneity parameters (e.g. compiler versions and hardware platform characteristics) [BCLO93].

Concerning the second question, we advocate an evolutionary path, thereby OMOS *fragments reside in traditional file systems, and constructed modules are represented via meta-objects in our* *persistent object store*. This provides logical uniformity, while easing application code migration from existing environments, and retaining applicability of the wealth of existing file management tools, e.g. source code control systems.

5 A Case Study: Modules For C++

Thus far, we have ignored the question of how our new conception of modularity can be reconciled with modularity notions built into existing languages. We are investigating this question through the application of OMOS to C++.

5.1 C++ As A Modular Programming Language

C++ programmers deal with two distinct notions of modularity: (i) *classes*, so fundamental to logical modularity, and (ii) *source files*, which deal with physical modularity. These two modularity dimensions share many characteristics, but have very different senses of composability, i.e. *inheritance* for classes, and *linkage* for files. Indeed, they are rather orthogonal in the minds of C++ software engineers, because (i) class definitions and source files rarely bear 1-1 relationships, and (ii) linkage is performed in a "class-less" universal namespace flattened by *name mangling*. In essense, we have two different spheres of intellectual endeavor at work: semantic relationships for classes, and software artifact management for files.

5.2 Fundamental Capabilities

We begin by considering how our module management approach can be applied to C++ in an evolutionary manner. That is, we will focus our attention on its utility for managing physical modularity, particularly in the presence of a persistent object store. In §5.3 we will intrude briefly on the class concept itself.

The utility of OMOS as a generalized linker and loader applies without qualification to C++ object files. Hence caching, sharing, interposition, and other capabilities are useful as described earlier — because C++ object code is indistinguishable from C code (except for telltale naming conventions). Indeed, under common C++ software engineering practice, there is little that OMOS can offer in the way of class manipulation, per se. When teamed with a persistent object store, however, important new roles arise. In particular, fully polymorphic load and store operations require that each stored object be associated with a C++ class *declaration* (layout) and *definition* (implementation). The class declarations are described by *dossier* objects, i.e. instances of system class dossier.

But what about class definitions? They comprise class method code, as emitted by a C++ compiler. The prospect of physically storing code in a persistent object store is unappealing on several grounds, including granularity, representational appropriateness, and duplication of effort.

Hence some means of indirect, *surrogate* representation is required. One might be tempted to use file names, but as remarked above, there is a poor correspondence in practice between individual files and individual classes. Our solution: represent class implementations as OMOS meta-objects (intensional modules), which use Jigsaw operations to (i) record a plan for building the class implementation from component fragments (files), and (ii) project out precisely the pertinent function and static data definitions.

5.3 Advanced Capabilities

The polymorphic reference capabilities of C++ make possible the manipulation of objects of subclasses unknown at compilation time [DSS90]. From our perspective, such *class hierarchy expansion* involves (i) recognition that class implementation code and static data must be dynamically linked and loaded, and (ii) ensuring that dossier objects are loaded for subsequent fully polymorphic manipulation.

One may also consider the more exotic possibility of changing the definition of a class on the fly (*class evolution*), e.g. by rebinding functions or adding members, or even altering the types of existing members [GS87]. In light of the restrictions sketched in §4.1, it is only reasonable to consider rebinding of C++ virtual functions, since C++ systems generally allocate virtual function tables in data segments, and hence remain *unfrozen*. This also applies to method function interposition.

Finally, we turn to the question of using our module manipulation capabilities more directly on the class hierarchy of a C++ program. While this may sound unwisely invasive, cogent arguments have been made for such higher level manipulation of class hierarchies [OH92]. Indeed, even the simplest OMOS functionality could have great utility in combining independently developed class libraries, e.g. name conflict resolution, rebinding to increase sharing, and module combination under structural typing rather than C++'s name-based typing. A host of provocative questions remain to be articulated and answered here.

6 Continuation Work

This approach to module management, while evolutionary in many respects, potentially has far-reaching effects on programming languages and software system architectures. In this section we examine briefly a few directions of particular importance and interest to us.

6.1 Applicability To Other Languages

We have sketched how the Jigsaw/OMOS modularity approach can be reconciled with, and augment, the class concept in C++. Clearly, a similar investigation is merited for influential genetically related O-O languages, notably Common Lisp Object System (CLOS), Smalltalk, Eiffel, Beta, and Oberon. Other important languages approach modularity from a more traditional program organization perspective, e.g. Modula-3 and Ada. Still others (Standard ML is the prime example) use advanced systems as a modularity foundation. The module systems in such languages should be studied for (i) reconstruction in Jigsaw/OMOS terms, and (ii) overall compatibility with our system server approach to module manipulation.

6.2 module As A Jigsaw ADT

The notion of an abstract data type representing modules pervades this paper. We have argued that such an abstraction arises naturally in program construction, and we have adopted it as the foundation of our design and implementation. In our case, the ADT is implemented by OMOS. Yet the module ADT is a useful concept independent of particular system architectures. Ideally, we should implement the ADT as a separate entity.

In a related effort, we are prototyping the module ADT as a *framework* [BL93] in Johnson's sense [JR91]. In this framework, notions such as modules and instances are represented by abstract classes, with specific module-based languages implemented via concrete subclasses. A client process like OMOS could extend or modify the framework to suit its particular needs. For example, the address loading constraints employed by OMOS are specific to linking, and would be added as an extension of the basic module ADT.

There are other valuable uses for such a framework. It may serve as a basis for the implementation of a family of interoperable language processors that share a common notion of object. This is not unlike the use of standard calling sequences to help ensure interoperability of procedural languages. Of course, a standard format for objects could be used by unrelated implementations as well. The advantage of the framework is that it implements the standard in a manner accessible to various compilers, saving implementation effort and encouraging adherence to the standard.

Another direction is to extend the semantics of C++. Jigsaw operators allow for richer behaviors than those of classes in C++. One example is the introduction of mixins [BC90]. The facilities introduced for *dynamically* applying Jigsaw operators allow inheritance at run time. These facilities could be employed by a modified C++ compiler, to support delegation. The difficult part here is deciding exactly how such features should be incorporated linguistically into C++; as described here, we already have an applicable implementation.

6.3 Type Issues

Our treatment of module typing is preliminary in several important respects. Firstly we have not as yet implemented the Jigsaw typing model in OMOS. As a result, typing is insecure, and type-sensitive operations are not provided. Full implementation of our typing model would open many attractive possibilities, such as type-specific linkage operations performing additional services such as data conversion across modules from differing base languages.

While typing in our model is structurally based, most existing languages use name based typing.

An argument can be made that structural typing scales better for highly intensive software reuse, but common software engineering practice relies on type names for application-specific semantic compatibility control (cf. derived types in Ada). An investigation must be made of how these two approaches can be made to productively coexist.

Finally, Jigsaw operators require precise typing of modules, e.g. to determine statically whether an attribute name clash will result when two modules are merge'd. Hence the Jigsaw module notion is only an approximation of an ADT in the fullest sense, which fundamentally relies on type abstraction [DT88, Bru93]. This implies that Jigsaw cannot be described in itself. The feasibility of Jigsaw-style combinators on modules based on true ADT's is a challenging open question.

6.4 Implementation

Finally, we are committed to the full development of a portable, comprehensive OMOS server. A prototype of an OMOS server has been implemented in C++, running on the Intel 386 platform under the Mach operating system, and the HP PA-RISC platform under both Mach and HP/UX[2]. Operating system dependent features have been isolated, thereby easing the port of OMOS to any system with modern virtual memory and inter-process communication facilities. As another example of its *value added* module management services, OMOS provides facilities for automatic monitoring of program execution, and transparent reordering of program binaries to improve program paging behavior [OMHL93].

7 Related Work

This effort builds on important work being done in many domains. We briefly survey two areas of particular relevance.

7.1 Module Systems For Individual Languages

As observed earlier, essentially all modern languages provide modularity notions of some form. We are particularly interested in those providing language-specific support for incremental module loading or combination. Two examples are:

- *Oberon:* The Oberon system [WG92], including the Oberon language [WG88], provides a highly integrated architecture for modules. Like OMOS, Oberon views programs as modules, and exploits sharing of previously loaded modules. Dynamic linking also plays an important role, as do module descriptors (akin to OMOS meta-objects). Modules have interfaces accessible by generic operations such as GetMod (name:ARRAY OF CHAR; VAR mod:Module) for accessing modules, and GetProc (name:ARRAY OF CHAR; mod:Module; VAR proc:PROCEDURE)

[2]HP/UX is a trademark of Hewlett-Packard.

for accessing procedures (methods) within modules. Single inheritance is provided in Object-Oberon [MT89].

- *Standard ML:* The Standard ML module system is among the most advanced in its exploitation of type-sensitive module manipulation [Mac86]. The basic building block is a *structure*, roughly akin to our module. A structure has a *signature*, which describes its typed interface. *Functors* map structure to structures, providing a function-like transformation notation. Functors can be used to achieve structure merging via repeated open directives. However, Standard ML structures cannot be abstract, so the framework-like development of module libraries is not supported.

7.2 Advanced Forms of Reflection

The OMOS approach to reflection may be called *stratified,* in that *modules* as objects (cf. C++ classes) and *module instantiations* as objects (cf. C++ objects) are manipulated by different languages and agents. This has the advantages of reuse of existing implementation support (e.g. C++ compiler and run-time systems), and (perhaps) reduced conceptual mystification.

However, separation of these levels inevitably introduces semantic barriers, inconsistencies, and missed opportunities. As a result, many researchers advocate an *introspective* approach to reflection, whereby meta- and object-level manipulations are performed within a single computational setting. Smalltalk [GR83] provides this in a modest way through its reliance on class objects. The most ambitious such language design is 3-Lisp [Smi82], which is predicated on reflection at an arbitrary number of levels.

The CLOS Meta-Object Protocol (MOP) [KdRB91] is a carefully designed introspective system with important practical applications (e.g. implementing persistence [Pae88, Lee92]). The MOP exploits several aspects of the CLOS O-O model, including generic functions, programmable method dispatch, and accessibility of class hierarchy information at run-time. These features permit much wider module manipulation capabilities than do the Jigsaw/OMOS model, but require support for reflection to be an integral part of the overall language implementation. Like OMOS, however, MOP implementation technology relies heavily on caching to improve performance for frequently used results.

8 Conclusion

We have outlined what we believe to be a fresh and attractive view of module management as a fundamental software system construction activity. Our conception of module is broadly applicable, and formalized in terms of an ADT. A partial implementation of that ADT exists in the form of a system process providing highly flexible linking and loading, as well as more value added services.

This approach has been conceived jointly by researchers deeply immersed in two specialties:

programming languages and system architecture. As such, we believe the result is a synergistic combination of insights and needs from both domains. From programming languages, we have taken inspiration from the power of inheritance in O-O languages, and the importance of types as a basis for secure, evolvable module interfaces. From system architecture, we have profited from the server process approach to system organization, and respect for the crucial role of module binding as a system flexibility governor.

Our implementation of this approach has not yet transcended the prototyping phase. Certain vital aspects remain to be addressed, notably non-trivial support for module typing. Hence many of our assertions remain largely unsubstantiated claims. We acknowledge at least two potentially controversial aspects of our overall endeavor:

Is it wise to move module manipulation to a language-neutral plane? Language-specific modularity mechanisms offer undeniable advantages, including integrated support for types, computational uniformity (more thorough reflection), and freedom from the backward compatibility constraints of *legacy code.* On the other hand, we see many disadvantages, including replication of module management services across languages, and inevitable operational incompatibilities resulting in unnecessarily *closed* systems. Is not connectivity with modules in standard programming environments (e.g. the X-window library) a practical necessity for any modern language implementation? If so, must this connection forever be done through the *thin wire* of C library calls? We see some hope for bridging this dichotomy by our framework-based approach, whereby a base language can delegate only some of its modularity management features to a system service, while retaining the remainder for overriding to achieve specific effects and policies. This presents the appealing prospect of module management systems occuping customizable intermediate points on the spectrum between fully *stratified* and fully *introspective* reflection.

Is our approach too "heavy weight"? Interaction with server processes inescapably incurs latency and semantic bandwidth limitations (although some ameliorations have been proposed [FLO93]). It is highly implausible that, despite our conceptually uniform view of objects, one would ever want to delegate run-time object construction to a server. Hence modules as objects will never be managed seamlessly with application objects. Does this matter? Perhaps it is only an implementation issue — but we conjecture that naming relationships will be the first stress point [OMK93].

Acknowledgements

We acknowledge the many contributions to this work by members of the Mach Shared Objects project, including Robert Mecklenburg, Mark Swanson, Robert Kessler, Jay Lepreau, Guru Banavar, Peter Hoogenboom, and Ravindra Kuramkote.

References

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

[BCLO93] Gilad Bracha, Charles F. Clark, Gary Lindstrom, and Douglas B. Orr. Modules as values in a persistent object store. Computer Science Department Technical Report UUCS-93-005, University of Utah, January 5, 1993.

[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.

[BL93] Guruduth Banavar and Gary Lindstrom. A framework for module-based language processors. Computer Science Department Technical Report UUCS-93-006, University of Utah, March 5, 1993.

[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.

[Bru93] Kim B. Bruce. Safe type checking in a statically typed object-oriented programming language. In Susan Graham, editor, *Proc. Symposium on Principles of Programming Languages*, 1993.

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

[DSS90] Sean M. Dorward, Ravi Sethi, and Jonathan E. Shopiro. Adding new code to a running C++ program. In *USENIX Proceedings C++ Conference*, pages 279–292. USENIX Association, 1990.

[DT88] S. Danforth and C. Tomlinson. Type theories and object-oriented programming. *Computing Surveys*, 20(1):29–72, March 1988.

[FLO93] Bryan Ford, Jay Lepreau, and Douglas Orr. Remote procedure call specialization with the OMOS object server. Unpublished paper, 1993.

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

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[GS87] Philippe Gautron and Marc Shapiro. Two extensions to C++: A dynamic link editor and inner data. In *USENIX Proceedings and Additional Papers C++ Workshop*, pages 23–34. USENIX Association, 1987.

[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.

[IL90] John A. Interrante and Mark A. Linton. Runtime access to type information in C++. In *USENIX Proceedings C++ Conference*, pages 233–240. USENIX Association, 1990.

[Jor90] David Jordan. Instantiation of C++ objects in shared memory. In *C++ at Work*, 1990.

[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.

[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.

[LK92] Gary Lindstrom and Robert R. Kessler. Mach Shared Objects. In *Proceedings Software Technology Conference*, pages 279–280, Los Angeles, CA, April 1992. DARPA SISTO.

[Mac86] David MacQueen. Modules for Standard ML. LFCS report, Dept. of Computer Science, Univ. of Edinburgh, Scotland, 1986. Part III of *Standard ML*, by Robert Harper, David MacQueen and Robin Milner.

[Mos92] J. Eliot B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18(8):657–673, August 1992.

[MT89] H. Mossenbock and J. Templ. Object-Oberon — a modest object-oriented language. *Structured Programming*, 10(4), 1989.

[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.

[OH92] Harold Ossher and William Harrison. Combination of inheritance hierarchies. In *OOPSLA Proceedings*, pages 25–40, October 1992.

[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 *Proceedings of the 26th Hawaii International Conference on System Sciences*, pages 232–241, January 1993. Also available as technical report UUCS-92-034.

[OMK93] Douglas B. Orr, Robert W. Mecklenburg, and Ravindra Kuramkote. Strange bedfellows: Issues in object naming under Unix. Unpublished note, July 1993.

[Pae88] Andreas Paepcke. PCLOS: A flexible implementation of CLOS persistence. In S. Gjessing and K. Nygaard, editors, *Proceedings of the European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, Berlin, 1988. Springer-Verlag.

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

[Smi82] B. Smith. Reflection and semantics in a procedural language. Laboratory for Computer Science TR-272, MIT, 1982.

[WG88] N. Wirth and J. Gutknecht. The programming language Oberon. *Software — Practice and Experience*, 18(7):671–690, July 1988.

[WG92] N. Wirth and J. Gutknecht. *The Oberon System.* Addison-Wesley, Reading MA, 1992.

Last revised July 26, 1993