A Framework For Module-Based Language Processors

Guruduth Banavar
Gary Lindstrom

UUCS-93-006

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

March 5, 1993

Abstract

A system composed of interconnected modules is a module-based system. We present an object-oriented (O-O) *framework* for the development of processors for module-based systems, such as compilers for O-O languages, linkers/loaders, and tools for user/system libraries. We claim that this framework, named *Jigsaw*, can reduce the development effort for such processors and also serve as a basis for interoperability among them. We address the issues of (i) how the abstractions in *Jigsaw* can be formulated as a framework, and (ii) how *Jigsaw* can be extended to construct processors for module-based languages, in the context of our prototype implementation in C++.[1]

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

1 Introduction

The development of processors for module-based languages and systems is a pervasive concern cutting broadly across programming language design and software engineering. However, progress in this area has come primarily through disconnected, language specific advances. We address this problem in its most general terms, by abstracting it to a language neutral plane. For the purposes of this paper, we informally define a *module* to be any software unit that provides a set of services as specified by its interface [Bra92]. One realization of the module notion is the *class* construct in O-O languages, where composition by inheritance is a crucial concept. Another is the *object file* produced by a compiler, with composition by *linking* operations. Private or shared system libraries constitute yet another example [See90]. Thus, compilers for O-O languages, system loaders/linkers, and library construction tools are examples of processors for module-based systems.

Even though such systems have differing views of modules, they share a common semantic goal that can be abstracted. We draw on recent work, *Jigsaw* [BL92, Bra92], that has succeeded in characterizing this commonality by formulating the basic operations on modules as a set of module *combinators*. *Jigsaw* is unusually powerful in accommodating differing views of modules. Bracha and Lindstrom [BL92, Bra92] have given a rigorous formal semantics for its notion of module abstractions, based on the work of Cardelli, Cook, Harper, Palsberg, Pierce, and others [HP91, CM89, Coo89, CP89, BC90]. For our purposes, an informal sketch of the semantics of *Jigsaw* will suffice (see Section 2).

In this paper we further develop the *Jigsaw* model, and implement abstractions extracted from *Jigsaw* as an O-O *framework*, in the sense of Johnson and Russo [JR91]. We overload the term *Jigsaw* to embrace this framework also. Processors for module based systems are implemented as *clients* of the *Jigsaw* framework that extend it in specific directions. As "proof of concept", we extend the framework to implement a simple applicative language and a simple imperative language. In subsequent sections we discuss in detail the issues arising when language processors are constructed using this framework.

Such a framework can serve many purposes, the most important of which are reduced system development time, and potential for interoperability. One directly foreseeable benefit is the development of families of O-O language processors that share a common notion of module as a *software substrate*. In the same sense that common calling sequences facilitate function-level inter-language linking, this approach can facilitate multi-lingual O-O programming [Har87]. In addition, we have found that the framework allows easy experimentation with language design, and has led to significant insights, especially regarding imperative language constructs, nested modules, and their interactions. Regardless of the other benefits of *Jigsaw*, the construction of coordinated language processors for significantly differing language designs by extending a single framework for modules is itself a novel application of O-O frameworks.

The interoperability potential of frameworks for module-based languages is being exploited in the *Mach Shared Objects* (MSO) project at Utah [LK92]. MSO is predicated on a broad view of modules, transcending particular O-O languages, and even the O-O notion itself. Instead, module manipulation is viewed as a software structuring issue that should be addressed in universal, system-wide terms. This viewpoint underlies *OMOS*, a programmable, dynamic linker and loader [OM92] which provides a language independent implementation of the abstractions in *Jigsaw*, as well as other enhanced system services [OMHL93], and forms a building block of the MSO persistent object store[BCLO93].

The next section presents *Jigsaw*'s view of modules, emphasizing the suite of module combinators it supports. Section 3 casts this viewpoint in framework terms, and examines the light it sheds on subtle issues such as the semantics of nested modules in an imperative client language. Our experience in prototyping *Jigsaw* as a C++ framework is also discussed. Section 4 outlines the steps in reifying this framework for particular client languages, both applicative and imperative. A basic familiarity with C++ syntax and semantics [ES90] is assumed in the presentations in these sections. Finally, future work is sketched, and our conclusions are summarized.

2 The *Jigsaw* View of Modules

An important modern notion of modularity is the *class* concept in O-O languages. Classes traditionally fulfill a variety of roles, including defining modules, defining subtyping relations, controlling visibility (via public/protected/private interfaces), constructing instances of a defined module, modifying and reusing existing program units via single inheritance, combining program units using multiple inheritance, resolving name conflicts, etc. Indeed, it was this observation that different O-O languages rely on different notions of *class* that led to the formulation of the central abstraction, *Module*, in *Jigsaw*. Such a formulation permits aspects of the class construct such as inheritance and visibility control to be "unbundled" as operations applied to modules. To this end, a suite of module *combinators* (i.e. operators) has been defined. In this section we informally introduce the semantics of modules and their combinators. For a detailed and formal treatment, the reader is referred to [Bra92]; a summary may be found in [BL92].

In *Jigsaw*, a module is simply a self-referential scope, associating labels (identifiers) with

```
module {
    x = 0; y = 0;
    dist = function(aPoint:{ x:Int, y:Int }) {
                sqrt(sqr((x - aPoint.x)) + sqr((y - aPoint.y)))
           }
} : { x:Int, y:Int, dist:{ x:Int, y:Int } → Real }
```

Figure 1: A module and its interface

meanings. These meanings can be *typed values*, bound through *definitions*, or simply *types* specified via *declarations* (defining a label subsumes declaring it). Declarations are used to create *abstract modules*, which can be manipulated but not instantiated. Modules do not contain any *free* references, i.e. references to labels that are not associated with any declarations, although (nested) modules may contain references to labels declared in statically surrounding modules. The semantics of nested modules are the focus of Section 3.3. Every module has an associated interface, which comprises the labels and types of all the visible attributes of a module. A surface syntax for a simple module and its interface is shown in Figure 1. Typing in *Jigsaw* is purely structural (sets of label-type pairs, without order or type name significance).

Our approach to characterizing modules involves three levels:

1. Module abstraction: This is *Module*, *Jigsaw*'s generic notion of modules.
2. Individual modules: These are particular module definitions, with specific labels, meanings and interfaces (e.g. C++ classes).
3. Module instances: Many languages support a notion of module instantiation, whereby objects are created from module definitions, with components determined by language-specific semantics (e.g. objects comprising non-static class members in C++).

To clarify how an abstraction such as *Module* can be reified in an O-O framework, consider Figure 2. In this figure we show a `class`[2] representing the *Module* abstraction with an interface consisting of methods representing module operations (or combinators, as mentioned above). It is important for the reader to understand the semantic intent of each of these

[2]The syntax and semantics of this `class` construct are not important at this point; just think of this as a generic O-O programming language.

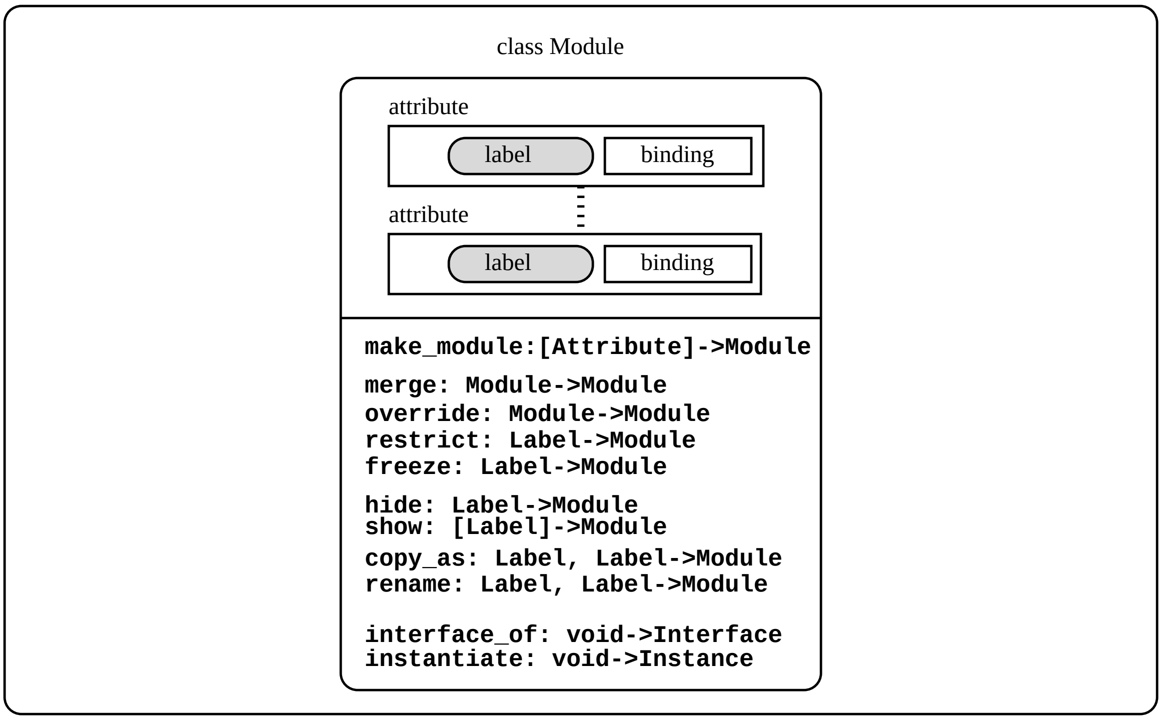

Figure 2: A first view of the *Module* abstraction

module combinators. In the following paragraphs, we informally describe the ways in which the *Module* abstraction models the many facets of conventional classes. This will set the stage for developing the framework characterization of *Jigsaw* in Section 3.[3]

Creation. A module M is created by invoking the module constructor function `make_module([Attribute])`[4] on a list of label-meaning pairs.

Instantiation. A *concrete* module `M` (i.e. one in which all labels have definitions) is instantiated by the expression `M.instantiate()`. The result of this expression is an *object* or *instance*. The module in Figure 1 can be instantiated to yield a point object with coordinates at the origin. Customized constructors and destructors can be modeled as methods explicitly defined within individual modules.

Combination. Two modules `M1` and `M2` may be combined using the `M1.merge(M2)` operation. The result is a new module in which all names declared in *either* `M1` or `M2` *or both* are declared, and all names defined in *either* `M1` or `M2` *but not both* are defined. Conflicting types or repeated definitions — name conflicts — are not permitted for

[3]The remainder of this section is a condensed extract of [BL92], Section 4.

[4]The syntax `[X]` stands for a list of data values of type `X`

a label. Note that `merge` does not provide any mechanism for resolving conflicts — other operators are used for this purpose. The `merge` operator is commutative and associative.

Modification. A module `M1` may be modified by another, `M2`, via an asymmetric operation `M1.override(M2)`, in which the attributes of `M2` override those of `M1`. If an attribute is defined by both modules, then the type of the attribute in `M2` must be a subtype of its type in `M1`, in which case the value from `M2` will appear in the result. Hence `override` provides a basis for dynamic function binding, as in C++ *virtual functions*. The `override` operator is associative and idempotent, but not commutative.

Name conflict resolution. A name conflict arising from merging two modules can be resolved in several ways. One can explicitly choose one of the conflicting definitions to prevail, using `restrict` (see below). This eliminates the conflict, but requires that one module's definition of the name to be relinquished, which may not be desired. Furthermore, the types of the conflicting attributes may be incompatible, in which case such rebinding is impossible. An alternative is to eliminate the conflict by renaming one label. This is always possible, and all attributes remain available. The renaming operator changes the label of a single attribute, i.e. `M.rename(a, b)` is equivalent to a textual replacement of all occurrences of the attribute name `a` in `M` by the name `b`. Attribute `a` must be at least declared by `M`, and `b` neither declared nor defined. One drawback is that in a structural type system, attribute names are meaningful for subtyping, and renaming may adversely affect polymorphism.

Attribute sharing. As mentioned above, `M1.merge(M2)` results in an error if both `M1` and `M2` provide a definition for a label. In contrast, if either `M1` or `M2` (but not both) define a label, the two usages are coalesced, as long as (i) a clashing definition has a type that is a subtype of the clashing declaration, and (ii) two clashing declarations have a subtype in common. Therefore declarations can specify sharing constraints among modules being combined, at the granularity of attributes. Such sharing is facilitated by the `restrict` operator. The effect of a restrict operation is to eliminate the definition of an attribute, but retain its declaration. It is not generally possible to completely remove an attribute from a module, because the module may contain internal references to the attribute. The `restrict` operator creates an abstract module, by making an attribute *pure virtual*. When several modules are combined via cascaded `merge`'s, sharing of conflicting attributes may be specified by restricting all but one. Any attribute being restricted must be defined by the argument module. The `restrict` operation is associative.

Restricting modification. The `M.freeze(a)` operation produces a module derived from `M` in which all references to `a` are statically bound, i.e. may not be stripped of the current definition of `a` via `override` or `restrict`. This provides a means for removing an attribute's subsequent redefinability, e.g. its *virtual* status in C++. As we shall see, `freeze` is often used in conjunction with visibility control.

Attribute visibility. Visibility control is implemented by the operations `hide` and `show`. `M.hide(a)` eliminates `a` from the interface of `M`. The attribute `a` must be defined by `M`. Conversely, `M.show(A)` hides all labels except those in list `A`. All attributes not listed in `A` must be defined in `M`.

Access to overridden attributes. Access to overridden definitions is supported via the use of the `copy_as` operator. `M.copy_as(a, b)` creates a copy of the `a` attribute, under the name `b`. The `a` attribute can now be overridden, while the old definition remains available under the name `b`. `M` must not already have declared an attribute `b`, but must have defined `a`.

The above summary is intended to serve as an introduction to the *Jigsaw* view of modules. However, there are deeper issues that deserve discussion, which we will examine as they arise. Many of these issues came into focus as a result of our effort to characterize *Jigsaw* as a framework. But first, we present this characterization.

3 *Jigsaw* As A Framework

An O-O framework [JR91] expresses the design of a software (sub)system in terms of objects and interactions between them, typically using a general purpose programming language. Frameworks are intended to capture the essential abstractions in an application domain, thereby allowing a developer to build applications efficiently by (i) specifying classes that inherit from classes in the framework and (ii) by *configuring*[5] instances of classes in the framework. Frameworks mostly comprise abstract classes, which are concretized by inheritance in an application. Frameworks thus promote design and code reuse through O-O concepts such as inheritance and polymorphism. Several frameworks have been developed for user-interfaces [Deu89, VL89, WGM88], and for many other domains as well [JR91]. In this section, we describe how the abstractions introduced in the previous section are reified as a framework for modules.

[5]Connection of objects from predefined concrete classes [JR91].

One way to exploit the *Jigsaw* model of O-O programming is to design new languages that embody this model. This direction is explored elsewhere [Bra92, BL92], and will not be treated further here. Another direction is to use *Jigsaw* to model and implement processors for existing languages. If suitable abstractions can be extracted from the *Jigsaw* model, they can be structured as a framework by associating a class with each of the key abstractions, thus allowing for reuse of design and code.

3.1 *Jigsaw* Classes

As suggested in Section 2, *Module* is the first obvious candidate for abstraction. This abstraction can be realized as a concrete `class Module`, providing each of the module combinators as a method. Similarly, the concept of an *interface* can be abstracted and realized as a concrete `class Interface` used to represent the interface of modules. At the *Jigsaw* framework level, only the ability to test two interfaces for equality or subtyping is postulated.

The concept of an *Instance* can also be abstracted and realized as a concrete `class Instance`. While modules are typically statically defined, *Instance* objects are constructed dynamically in most languages. As mentioned earlier, an instance is created via the `instantiate()` operation on a *Module*. However, an instance does not support the same operations as a module. In fact, the key method that `Instance` must provide is `select`, which when supplied a label, returns its binding. The `select` method corresponds to the notion of sending a message to an instance, and encapsulates the functionality of determining the exact binding to return. The latter can be implemented in several ways, but the important point is that the framework determines a common logical composition for instances and a mechanism by which to use that composition. Furthermore, a client language processor or a client language program may need to determine the type of an instance. This can be achieved by accessing the instance's module via `module_of()`, and then by invoking the `interface_of()` method of that module. While the notions of a module's *interface* and an instance's *type* are conceptually distinct, the above approach to determining an instance's type represents our preliminary view that the type of the instances of a module can be implemented as the interface of the module.

The above abstractions are defined relative to the notions of *value*, *type* and even *label* in a client language, L_c, over which *Jigsaw* abstracts. L_c must provide its own concept of values, types and labels. Thus, *Value*, *Type*, and *Label* are incompletely specified abstractions within the *Jigsaw* model, and are therefore specified as abstract classes `Value, Type`, and `Label`. *Jigsaw* only requires the *Label* abstraction to supply a notion of label equality via the method `label_eq(Label)`. The *Value* abstraction is required to return its *Type* when queried with `type_of()`. *Type* in turn must supply notions of type equality (`type_eq(Type)`) and subtyping (`subtype(Type)`). A particular modular programming language is implemented by supplying definitions for these methods in their associated abstract classes, and possibly by extending the functionality of abstractions, or by adding other abstractions. These definitions and extensions constitute an implementation of L_c.

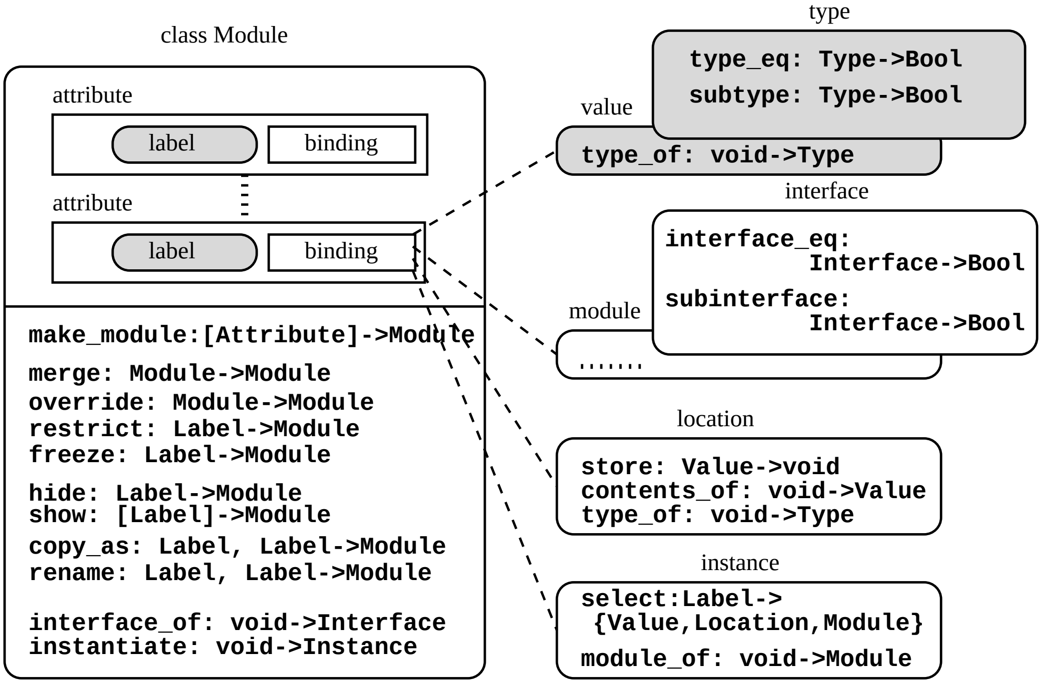

Figure 3: An overview of the *Jigsaw* framework

If each of the abstractions mentioned above is realized as a class, we have a framework that can be pictured as Figure 3. There are abstractions in the figure other than the ones mentioned above — these will be explained later in this section. Each box stands for an abstraction, with shaded boxes standing for incompletely specified abstractions (corresponding to abstract classes) in the framework, and names in lower case letters (e.g. `module`) standing for instances of classes with the same names starting in upper case (e.g. `class Module`).

External clients [Deu89] of `class Module` invoke the constructor `make_module` with a list of attributes, each of which is a label-binding pair. We generalize the notion of *binding* in *Jigsaw* to include not only values and nested modules, but also *declarations*, i.e. types and module interfaces. Thus, `class Module` is expected to be used by clients by "configuring" each instance of it, and we expect it to be rarely subclassed.

A module's interface can be obtained by invoking the `interface_of()` method. Invoking the `instantiate()` method on a concrete module returns an instance of it, which is created by sharing module-level bindings, and copying instance-level bindings (see Section 3.2). Such module instances are objects of `class Instance`, which implements the notion of instance described earlier.

`Value`, `Type`, and `Label` are abstract classes, and implement the corresponding notions. Classes `Attribute` and `Binding` are supporting abstractions, respectively implementing a list of attributes each of which is a label-binding pair and a container for our notion of binding described earlier. The class `Location` was added later to the framework after the initial implementation, as we gained more insight into the language modeling power of *Jigsaw*. This abstraction, and other subtleties of the *Jigsaw* approach, are described in the following sections.

3.2 Applicative vs. Imperative Semantics

Module operations are all applicative in nature, i.e. they map modules to modules, without side-effects. However, the client language being modeled may be applicative or imperative. Which it is has significant implications on the semantic refinement of abstractions in *Jigsaw*, especially in the case of nested modules. This section highlights these implications in preparation for the next section which discusses in detail the semantics of nested modules.

In an applicative L_c, one may be tempted to equate a concrete module and all its instances. However, there are compelling reasons to distinguish the two. Apart from the fact that modules and instances are fundamentally different entities (e.g. support different suites of operations), instances may be used polymorphically in L_c (e.g. as parameters to functions) whereas modules may or may not. Moreover, in the case of an imperative L_c, a concrete module and its instances are generally distinct due to the presence of references to non-local bindings shared in a surrounding scope (see Section 3.3).

When *Jigsaw* is used to model an applicative language, modules bind labels to either values in L_c (constants) or to other (sub)modules. In contrast, a label in an imperative L_c can also be bound to a *location*, following the standard denotational model of imperative languages with stores [Gor79]. Hence we must provide an abstraction of this notion in *Jigsaw*. A location can hold a *storable value*, the exact definition of which is client dependent. The set of entities that comprise storable values form the set of *first class values* for a particular L_c. It seems reasonable to expect storable values to include at least *values* in L_c, but could possibly also include instances, locations and even modules. It is common for O-O programming languages to allow "slots" that can contain instances and pointers to instances — these can be modeled with a *Location* abstraction.

Location bindings to labels can be either *module-level* (e.g. `static` in C++), where the same binding is shared by all instances of the containing module, or *instance-level*, where each instantiation of the containing module results in a new location being bound to the label. In our current prototype implementation of the framework, only instance-level location bindings are supported, but it can be extended easily to accommodate module-level bindings also. Note that this distinction of module-level and instance-level is necessary only for location bindings, and not for other types of bindings (e.g. value, module), and hence does not arise in the applicative case.

Jigsaw supports an implicit notion of *self*, i.e. local attributes may reference each other within a module. An explicit notion of self brings with it typing issues relating to preservation of encapsulation and feasibility of separate compilation. These have been explored in the literature [HC90, Bru93, Bra93], but not addressed by the framework in its current formulation.

3.3 Nested modules

Nested modules are an important requirement for modularity because, while enhancing name space separation, they also provide a basis for hierarchical module development. In addition, this provides an integrated notion of overall program structure — running a program is simply instantiating the *top-level* module and invoking a user-written initialization method.

For example, a framework itself can be regarded simply as a module, with ordinary classes as nested modules, and the process of extending the framework can be viewed as extension of the module by merging, sharing, hiding, renaming, etc. This capability is vital to programming in the large, as C++ application developers are ruefully aware (since all C++ class and attribute names are flattened by *name mangling* into a single name space). As noted above, modules allow for data sharing among its instances via module-level attributes. Another form of sharing can be achieved through nested modules, which can produce instances of nested modules with non-local references sharing access to definitions in common surrounding scopes. The surrounding scope thus serves as a shared data repository for these instances, and helps achieve sharing semantics such as that provided by `static` member data in C++. Nested modules could also serve as the basis for other interesting possibilities such as combination of inheritance hierarchies [OH92].

It was noted in Section 2 that modules can contain references only to attributes that are declared either locally (in the same scope) or non-locally (in some surrounding scope). In the original definition of *Jigsaw* in [Bra92], non-local references to surrounding *dynamic* scopes are modeled using standard denotational techniques, by having an object denote a dynamic environment which is passed as an argument to the module constructor function. In our prototype implementation, this mechanism is implemented using another standard technique, by having a nested module access its surrounding scope via a pointer to the dynamically surrounding `instance` object.

In addition to the above two models of non-local reference, we propose a third model, one we believe could facilitate application of the *Jigsaw* approach to, for example, *dynamic loading* and *object promotion*. This view, presented below, is characterized by the introduction of *instance-level* operators corresponding to the module combinators.

In this model, non-local references are handled by endowing nested modules with implicit declarations of non-local bindings. When a nested module is `select`'ed, the definitions bound to these non-local references are *imported* from the surrounding (dynamic) scope. The import operation is broken down as follows: `show` the non-locally referenced labels from the surrounding scope, then `merge` the resulting module with the nested module containing the references. Note that this `merge` operation cannot cause a conflict because definitions for non-local references cannot exist within nested modules — otherwise they would be local references. Furthermore, the fact that non-local bindings are imported from a *dynamic* surrounding scope has the salutary benefit of guaranteeing that the imported label binding always has a definition, and not simply a declaration (since instances cannot be abstract). We note, however, that the original *Jigsaw* formulation cannot accommodate this definition importation effect with the use of module operators, and we thus introduce corresponding *instance level* versions of module operators.

A subtlety concerning the import of non-local bindings is the possibility of conflicts (e.g. when `merge`'ed) between nested modules importing the same label (and its binding) from a common surrounding scope. This can be solved, however, by `hide`'ing all imported bindings immediately after importing them. The `hide` operation statically binds the imported definitions, and then removes their labels from the nested module interfaces. This preserves the interface of the nested module as it was before the import, and ensures that subsequent `merge` conflicts cannot arise as a consequence of the the import operation. Moreover, we observe that the import of a location binding can be designed to either retain the module-level or instance-level ("static" or "dynamic") nature of the original declaration, or to be subject to explicit programmer control. In the former case, if the location was originally declared to be instance-level, all nested modules that reference it will share the same binding, but instances of a nested module will each be allocated a new location binding and hence will not share the location. If the location was originally declared to be module-level, then all

```
class Module {
  Attribute* attrib_list;
  Instance* parent;         // Link to surrounding scope
  /* ... */                 // Other private data
public:
  Module (Attribute*);      // Constructor
  Module* merge (Module*);
  Module* override (Module*);
  Module* restrict (Label*);
  Module* freeze (Label*);
  Module* hide (Label*);
  Module* show (Labels*);
  Module* rename (Label*, Label*);
  Module* copy_as (Label*, Label*);
  Interface* interface_of ();
  Instance* instantiate ();
  /* ... */                // Other utility functions
};
```

Figure 4: C++ implementation of *Module*

nested modules that refer to it *and* all instances of such nested modules will share the same location.

3.4 A C++ Prototype

It is fairly straightforward to translate the *Module* abstraction into a C++ class (see Figure 4). Each method in the public interface implements the corresponding module combinator introduced in Section 2. If an instance of `class Module` represents a nested module, the private slot `parent` points to the instance that contains it, as described in Section 3.3. In the current prototype, `class Attribute` has been implemented as a simple linked list of label-binding pairs, with operations to add, remove, find, etc. such pairs. When the `instantiate` method is invoked, an object of `class Instance`, with its own copy of instance-level attributes, is returned.

The implementation of module combinators deserves some discussion. Implementing `merge, override`, and `restrict` is fairly straightforward. In order to implement the `freeze` method, we must first implement the notion of self-reference within a module. The binding for an attribute might be via a module's self-reference to a sibling attribute, which might not be defined yet — in which case we do not have a binding at module-definition time. Consequently this notion of self-reference to attributes must involve some form of delayed binding, which we capture with the help of another abstraction, implemented as `class Reference`. This class provides a `dereference()` method that can be used to retrieve bindings of labels. Hence `class Reference` provides a level of indirection in accessing self-referenced attributes that could potentially be rebound. This is a module-level analogue of object-level dynamic binding as implemented by virtual function tables in C++. In both cases, the indirection is primarily motivated by code reuse and separate compilation.

Thus, the `freeze` method is implemented so that it statically (i.e. at module-definition time) `dereference`'s the binding of its argument. The `hide` method is implemented in terms of `freeze` and `restrict`. Implementing `rename` and `copy_as` is straightforward. Classes `Label`, `Value`, `Type`, and `Location` are implemented almost directly as described earlier, as C++ abstract classes.

3.5 Client Syntax and Semantics

As formulated above, a client will use *Jigsaw*'s abstractions by first creating objects of `class Module` (via the `Module` constructor), then by invoking `instantiate` on the module objects thus obtained to create objects of `class Instance`. So, all *Jigsaw* related entities in the client are instances of C++ classes, but are interpreted semantically differently depending on the C++ class of which each is an instance. Thus, in our current prototype, all client language processors are written in C++, and all client languages have C++ surface syntax. That is to say, each of the client languages is really C++ augmented with the *Jigsaw* model of modules, which is provided as a set of classes.

However, we can permit clients to have their own surface syntax by adding *parse* methods for each of the abstractions in *Jigsaw*. For example, we can have an abstract method `parse_module: Stream->Module`, that produces a module object given a *Stream* (as defined by the client) of characters. A default implementation of this method that implements one particular syntax for modules will be provided in the framework. This method is an *abstract* method since it constructs a parse tree for the given stream by calling parse methods of other classes, e.g. `parse_value, parse_type,` and `parse_label`, which are expected to be provided by the client. It would also recognize denotations of module combinators in its input, and call the appropriate method that implements each such combinator. Given the `parse` methods, a parser for a client with its own syntax is built by using the parse methods provided, and by supplying parse methods that are not. Typically, however, we expect that clients will redefine all the parse methods to support their own syntax. We look forward to the day when O-O languages will directly support I/O on customized lexical representations for programmer defined data structures, as do some functional languages (e.g. CAML).

4 Language Processors Based On The *Jigsaw* Framework

This concludes our general discussion of the abstractions in *Jigsaw*. In this section, we illustrate how these abstractions can be used to develop processors for a simple applicative language and a simple imperative language, both with C++ surface syntax. The languages support the creation and manipulation of *Jigsaw* modules, and the creation and use of instances of these modules. We will not introduce any new surface syntax for clients here (i.e. no parse methods will be used), and thus the client syntax may at times seem a bit baroque.[6] In the following section, we will first analyze the process of building language processors by extending *Jigsaw*.

4.1 Extending *Jigsaw* To Build Processors

In Section 2 we enumerated the three levels of module characterization underlying the *Jigsaw* approach. We now clarify and refine those levels, in the context of an explicit client language L_c. We also offer some observations about the artifacts (objects) arising at various levels.

1. [MODULE ABSTRACTION:] This is `class Module`, the class representing *Jigsaw*'s notion of modules. `Module` is concrete, because it includes a generic definition of all its attributes. However, it remains indirectly abstract, since it relies on abstract auxiliary classes, as shown in Figure 3.

2. [MODULE IN L_c:] The *Jigsaw* notion of modules tailored to L_c is defined by providing concrete definitions of these auxiliary classes, or by subclassing `class Module` in order to refine or customize it, as appropriate for L_c modules.

3. [INDIVIDUAL L_c MODULES:] Once the L_c notion of modules is made complete, particular L_c modules can be defined, with specific interfaces, labels and bindings. These are obtained by invoking the `Module` constructor method of `class Module` as refined in L_c.

[6] Although certain extensibility features of C++ such as operator and function overloading enable a surprisingly readable surface syntax.

4. [L_c MODULE INSTANCES:] Finally, if the concept is supported by L_c, instances (objects) derived from particular L_c module definitions can be created by invoking the `instantiate` method of an individual L_c module.

This four-stage reification process is central to exploiting the *Jigsaw* approach to managing modules. It is crucial for the reader to understand the role of each level, and to maintain their conceptual separation. Nevertheless, when *Jigsaw* is represented as a framework in a single O-O language (e.g. C++, as in Section 3.5), a very beneficial representational flattening occurs. In particular, stages (3) INDIVIDUAL L_c MODULES and (4) L_c MODULE INSTANCES are *each* represented as objects in the framework implementation language (e.g. C++). The objects representing (3) automatically constitute *dossiers* in the sense of Interrante and Linton [IL90]. These can drive fully polymorphic level (4) object manipulation functions, such as storage and retrieval from a persistent object store. Indeed, dossiers can be associated with (2) objects as well, capturing implementation properties shared by all L_c modules, e.g. dispatch table layout conventions. The implications of this representational uniformity advantage of *Jigsaw* frameworks is explored in [BCLO93].

4.2 An Applicative Language

We now present a simple applicative language implemented by extending *Jigsaw*. In this L_c, modules are created by invoking the `Module` constructor and specifying a list of labels and their bindings, which can be either values, types, (sub)modules, or interfaces. Such labels are instances of `class Label_Lc : public Label`, which implements labels in L_c as, for example, strings of characters, and provides an implementation for the virtual method `label_eq`. Similarly, values are instances of `class Value_Lc : public Value`. The `class Value_Lc` concretizes the *Jigsaw* notion of *value*, by implementing the domain of computable values in L_c. Let us suppose that our L_c provides integers, floats, characters and functions as value bindings for labels. The function-valued bindings correspond to *methods*. `Value_Lc` must also provide implementations for *operations* on the primitive values, e.g. arithmetic on integers and floats, and *application* of method functions. When queried for its type, a `Value_Lc` object must return an instance of `class Type_Lc : public Type` that implements the space of types in L_c. `Type_Lc` also provides implementations for virtual functions `type_eq` and `subtype`.

A simple module definition in this applicative language is shown in Figure 5. In this fragment of C++ code, the variables `x, y, z, o,` and `n` are first initialized to instances of `class Label_Lc`. Within `class Label_Lc`, the operator "`=`" has been overloaded to take any binding (e.g. an integer, or an instance of `class Value_Lc` or `class Module`) as its argument

```
Label_Lc x, y, z, o, n;
Module* m = new Module
  (x = 13,
   o = self_refer(n)->merge
          (new Module (y = 3.1415)),
   n = new Module (z = non_local(x))
   );
```

Figure 5: Example module in an applicative language

and return an instance of `class Attribute`. Thus, the C++ expression "`x = 13`" in the figure actually invokes the overloaded "`=`" operator of `class Label_Lc` with 2 arguments — the `Label_Lc` object in variable `x`, and the value `13`, and produces an instance of `class Attribute` that represents the "binding" of the label in `x` to the value `13`. Several such `Attribute` instances are passed as arguments to the constructor for `class Module`. The auxiliary function `self_refer(n)` returns an instance of `class Reference` that contains a link to the binding of label `n`. The function `non_local` implements the functionality of importing bindings from surrounding scopes, as described in Section 3.3.

Module `m` can be instantiated using, say, `Instance* i = m->instantiate()`. The value bound to label `x` can then be accessed using `i->selectValue(x)`. Instances of the nested module bound to the label `n` can be created using `i->selectModule(n)->instantiate()`, and so on.

4.3 An Imperative Language

For an imperative language, we must provide a *store* consisting of locations each capable of holding a storable value. In this L_c, let us suppose that values (i.e. instances of `class Value_Lc`) are the only types of entities that can be stored into a location. If we wished to provide for storing instances (i.e. objects of `class Instance`) or other entities, we would need to subclass `class Location` accordingly. We allow module attributes to be bound to locations, i.e. instances of `class Location`. `Value_Lc` objects can then be `store`'d into and retrieved from (using `contents_of`) `Location` objects.

An example of a module definition using such objects is shown in Figure 6. The L_c implemented in this illustration treats location bindings as instance-level (i.e. non-static, see Section 3.2) by default. Integer values can be `store`'d and retrieved (using `contents_of`) from the location binding of `x`. The non-local reference to label `x` within the nested module

```
Label_Lc x, w, z, n;
Module* m = new Module
  (x = new Location (66),
   n = new Module (w = ...non_local(x)...),
   z = 96.8
   );
```

Figure 6: Example module in an imperative language

bound to n results in an import of the binding for x. The import operation is implemented in the prototype to treat imported location bindings as module-level. This allows the contents of the imported location binding to be shared among all instances of the nested module bound to n, whereas each instance of the outer module gets a new location binding for attribute x.

4.4 C++ As A Framework Implementation Language

We note that it is desirable for an O-O language to support the following features to maximize its utility as a framework specification language: (i) guarantee monotonic extension of interfaces by subclassing, especially if classes and types are coupled, (ii) prevent leakage of encapsulation, (iii) provide run-time type information, and (iv) support multiple inheritance. Extra expressive power in the language, such as the ability to specify invariants [JR91] would further enhance its utility for framework implementation.

Run-time type information becomes desirable for the following reason. Consider an abstract class A which has a single public pure virtual method `void foo (A*)`. The intention is that a concrete subclass of A, say `class B : public A`, will concretize the foo method and perhaps add its own private data to do so. But the implementation of foo in B can only view its parameter as a pointer to A, although in reality it will be an instance of some concrete subclass of A, perhaps B itself! Hence foo does not have access to the private data of its parameter, unless it is downcasted to a known concrete subclass of A. In general, it might not even be possible to know which concrete subclass of A the parameter points to an instance of — hence run-time types become essential for safe downcasting. We note that such safe downcasting is already supported in *Jigsaw* at the module level by the module_of link in class Instance objects. However, the problem remains in the C++ framework implementation — although dossiers solve this problem in the MSO object store [BCLO93]. An alternative would be to allow the programmer to specify the type of a function parameter as the type of `this` (corresponding to the bound variable *MyType* in [Bru92]), but other typing problems accompany this approach. Yet another alternative would be to provide a default or *canonical* implementation for the private data of `A`, but this is clumsy and requires inordinate foresight by the framework designer. All this, of course, is related to ADT-style of programming which is only partially supported by C++ (more on this in Section 6).

In our experience, C++ as a framework implementation language has scope for improvement. Here are some of the shortcomings we observed during this implementation effort: (i) the lack of run-time type information, (ii) the restriction that overloaded functions cannot be distinguished simply by return types, and (iii) the requirement that all definitions of a virtual method must match exactly in type signature. Although we are respectful of the engineering judgments that entered into the design of C++ [ES90], we nevertheless observe that its utility as a framework specification language is adversely affected by these shortcomings.

5 Related Work

The *Jigsaw* framework approach to building language processors has a strong relation to *reflective systems*, and is very similar to languages with meta-object facilities such as the CLOS MetaObject Protocol (MOP) [KdRB91], and Smalltalk-80 [GR83] metaclasses. The CLOS MOP supports user-redefinable protocols for meta-objects such as class, instance, generic function, method, etc. CLOS MOP provides the basis for the development of a "space of languages with the default language being a distinguished point in the space." Smalltalk-80 provides a highly intertwined collection of meta-classes starting from `class OBJECT`.

Nevertheless, there are important differences between our approach and previous ones. Our notion of modules is motivated by a desire to uniformly treat the semantics of inheritance. In addition, *encapsulation* is an important semantic requirement in *Jigsaw*, since we believe that it is crucial for software development in the large. Static typing is another important consideration in *Jigsaw*. Furthermore, the *Jigsaw* class interfaces are derived from a rigorous semantic foundation, rather than the requirements of dissenting language designs already in existence. The *Jigsaw* framework finds applications in the interoperability among languages, linkers and libraries. As noted in the next Section, the *Jigsaw* framework can be used for many purposes that the CLOS MOP has been put to use, notably persistent objects.

6 Future Work

We are encouraged by our preliminary results in this framework design and prototyping exercise to envision further investigation in several areas.

1. *Module-based language processors:* Clearly, we are keen to determine the practical feasibility of refining our prototype into a *software breadboard* for experimentation in constructing genuinely useful processors for module-based languages. The integration of *yacc/lex*-based tools to define *parse* methods for surface syntax would greatly aid in hiding C++ syntax from disinterested test users. Such a full-fledged implementation could provide an excellent context for experimentation in fast and adaptive method lookup implementations [HO92, CDMB89]. The design of a realistic language processor for a modular extension of the programming language C encompassing these ideas is currently in progress.

2. *Implications of persistence:* Persistent stores raise many object module management questions, including interoperability of O-O language processors [Mec91], transaction control as an inheritance concept [Frø92], class evolution [DSS90], and instance level module operations for *object promotion* [GS87, Sta91]. We believe our *Jigsaw* framework prototype will prove very useful for carefully exploring these issues within the context of a realistically complete, yet malleable, concept of modules.

3. *An ADT-based Jigsaw:* In Section 4.4 we commented on the difficulty of accessing private implementation data via abstract classes, and raised the possibility of default or canonical data representations. A better longer term approach would be to redevelop the *Jigsaw* framework within the context of genuine abstract data types, with existential types that permit tracking of hidden representation types via *witness types* [CW85, DT88]. To quote Bracha [Bra92]:

   > "A formulation [of *Jigsaw*] based on existentially quantified types is problematic, because of type abstraction. In particular, creating new abstract data types by combining the abstract types from two modules runs into the same difficulty that has arisen time and again in this dissertation — how to type-check inheritance in the presence of type abstraction. A rigorous definition of inheritance on ADTs is an important and substantial research issue."

7 Conclusions

We have advanced the idea that it is feasible and worthwhile to abstract the notion of module, and cast that abstraction into an O-O framework called *Jigsaw*. This idea has been explored in the concrete representational context of a C++ based prototype, within which we have implemented two simple module-based languages. This experiment has confirmed our belief that characterizing *modularity* in terms of a *framework* strengthens our understanding of both concepts. In particular, our *Jigsaw* prototype has enabled us to articulate and explore subtle areas where the semantics of fairly well understood concepts interact in surprising ways, notably nested modules and imperative client languages. Incremental refinement of the original *Jigsaw* conception has also occurred through our experimentation. In addition, light has been shed on requirements for O-O languages for implementing such frameworks, notably C++. Several areas of attractive future work remain, including the construction of genuinely useful processors for module-based languages, integration of support for persistence into the *Jigsaw* framework, and reformulation of *Jigsaw* in abstract data type terms.

Acknowledgements

We are indebted to Gilad Bracha for his fundamental work in conceiving *Jigsaw*, his generosity in permitting us to build on one of his unpublished working drafts, and his detailed comments on a later draft of this paper. The insights and support of Charles Clark, Douglas B. Orr, and all other MSO project participants are also gratefully acknowledged.

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.

[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.

[Bra93] Gilad Bracha. Private communication. Electronic mail, January 28, 1993.

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

[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.

[CDMB89] R.C.H. Connor, A. Dearle, R. Morrison, and A.L. Brown. An object addressing mechanism for statically typed languages with multiple inheritance. In Norman Meyrowitz, editor, *OOPSLA '89 Conference Proceedings*, pages 279–285. ACM Press, 1989.

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

[Coo89] William Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

[CP89] William Cook and Jen Palsberg. A denotational semantics of inheritance and its correctness. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 433–444, 1989.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

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

[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.

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

[Frø92] Svend Frølund. Inheritance of synchronization constraints in concurrent object-oriented programming languages. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, LNCS 615, pages 185–196, Utrecht, The Netherlands, July 1992. Springer-Verlag.

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

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

[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.

[Har87] W. Harrison. RPDE^3: A framework for integrating tool fragments. *IEEE Software*, 4:46–56, November 1987.

[HC90] Jin Ho Hur and Kilnam Chon. Self and selftype. *Information Processing Letters*, 36:225–230, 1990.

[HO92] William Harrison and Harold Ossher. Attaching instance variables to method realizations instead of classes. In *Proc. International Conference on Computer Languages*, pages 291–299, San Francisco, CA, April 20-23 1992. IEEE Computer Society.

[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.

[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.

[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.

[Mec91] Robert W. Mecklenburg. *Towards a Language Independent Object System*. PhD thesis, University of Utah, Salt Lake City, Utah, June 1991.

[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.

[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.

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

[Sta91] Manfred Stadel. Object oriented programming techniques to replace software components on the fly in a running program. *ACM SIGPLAN Notices*, 26(1):99–108, January 1991.

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

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