Modular Language Processors As Framework Completions

Guruduth Banavar
Gary Lindstrom

UUCS-93-026

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

October 21, 1993

Abstract

The conceptual and specificational power of denotational semantics for programming language *design* has been amply demonstrated. We report here on a language *implementation* method that is similarly semantically motivated, but is based upon object-oriented design principles, and results in flexible and evolvable language processors. We apply this technique to the area of object-oriented (O-O) languages, in the form of a general metalevel architecture for objects and inheritance that facilitates the development of compilers and interpreters for O-O languages. This development strategy maintains architectural modularity by mapping conceptual language design decisions to isolatable parts of resulting language processors. Our architecture, which is presented as an O-O *framework*, is characterized by (i) support for a broad set of modularity features including encapsulation and strong typing, and (ii) an "unbundled" view of inheritance, semantic features of which are decomposed by means of a set of module combination operations (combinators). We describe an implementation of our framework in C++, and assess its utility by constructing a compiler for a simple O-O extension to the programming language C. We further argue the flexibility of the resulting processor by outlining the incorporation of several significant extensions to the basic module language. We claim that the use of such a framework for compiler construction has many advantages, including a systematic language development method, processor software reuse, language extensibility, and potential for interoperability among languages.[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 denotational approach to programming language design employs abstraction to specify properties that a language's implementation must satisfy. The conceptual and specificational power of this approach is widely acknowledged [AW82]. However, the direct realization of language processors embodying a denotational semantic specification in modular and evolvable form remains an elusive research goal [JS80].

One may ask whether alternative semantic formulations might more effectively bridge this gap between language design and implementation. Recently, concepts fundamental to object-oriented programming have been successfully specified in a formal semantic manner [Coo89, Bra92]. Moreover, object-oriented programming and design techniques have matured through extensive application in a variety of areas. We capitalize upon both of these advances, and propose a metalevel O-O architecture for an *object system* (i.e. an object model and associated inheritance semantics) that is suitable for modeling and building processors for a wide range of O-O languages.

Our architecture does not encompass all aspects of a comprehensive language design — for example, it does not prescribe control structures, or the base computational value domain. Rather, we provide key abstractions and semantic structures characteristic of an object system constituting a starting point for the language developer. We also seek to make our object model "open-ended", so that it can used in a variety of situations. This is achieved (i) *conceptually* by "unbundling" the class concept and inheritance semantics traditionally found in O-O languages, and (ii) *implementationally* by formulating our model as an *O-O framework* [JR91] that uses *abstract* classes, i.e. incompletely defined classes that are completed by the framework user. The framework user can either adopt the default semantics of the object system, or refine it to suit particular language requirements.

The advantages of using such a framework for developing compilers are many, and accrue both as a result of object-orientation and our specific metalevel architectural design. As a result of object-orientation, we seek the well-known advantages of (i) software reuse and consequently reduced development effort, and (ii) design extensibility, i.e. incremental refinability of abstractions. Other advantages include:

- *Language extensibility:* the ability to layer object systems on legacy non-object languages;
- *Evolvability:* the ability to easily model existing forms of inheritance, hence to re-engineer compilers for existing languages, thus facilitating further evolution, e.g. adaptation of new advances in type checking [BG93], or combination of inheritance hierarchies [OH92]; and
- *Experimentation:* the ability to refine and combine the framework abstractions in flexible ways to create new and interesting object models and to investigate the resulting feature interactions.

We illustrate these advantages in Section 4, where we describe the construction of a processor for an O-O extension to the programming language C as a completion of our framework. A further advantage of our approach that we are currently investigating is the potential for multi-lingual O-O programming. We expect this to be the result of logical compatibility of objects across independent extensions of the framework, in the same sense that common calling sequences facilitate function-level inter-language linking [Har87].

Traditionally, abstraction and reuse in compiler construction have been fruitfully applied to implementation data structures, e.g. symbol tables, parse trees, etc. It seems natural to extend these advantages by abstracting concepts in the *computational domain* of the language being implemented. It is crucial, however, that the identified abstractions be carefully conceived, in order to maximize their range of applicability. We believe that we have identified such a set of abstractions, broadly applicable to object-oriented languages and systems.

In formulating our object architecture, we have realized that the notions central to O-O programming, such as the *class* construct and composition by *inheritance*, are applicable not only to O-O languages, but to a range of module manipulation systems as well. For example, *object files* produced by a compiler are composed by *linking* operations. Indeed, we have reused the same framework abstractions as a basis for a programmable linker/loader [OM92]. Private or shared system libraries constitute yet another example [See90]. Hence, we have found it advantageous to address this problem in its most general terms, by abstracting it to a language neutral plane.

It has been recognized in the past that object-oriented languages and systems are themselves perfectly suitable domains for object-oriented design. The idea is to model notions fundamental to O-O programming, such as *class*, themselves as objects. This has been exploited in the design and construction of O-O languages and environments, such as Smalltalk [GR83] and the CLOS Meta Object Protocol [KdRB91]. However, such metalevel architectures are tightly coupled with their base languages within highly dynamic and reflective environments. While this coupling enhances application development flexibility, it causes the metalevel architecture to be too restrictive for full-fledged language and system development. Moreover, it is difficult to untangle these metalevel architectures from their linguistic environments for separate reuse.

Our metalevel architecture, in contrast, focuses on a broader range of of concerns, including the development and exploration of new O-O languages as well as providing a basis for object management system services. In particular, we are strongly motivated by modular construction of compilers for O-O languages. In presenting our architecture, we first describe our object model followed by a framework formulation of it. We then describe the development of an experimental object-oriented extension to the programming language C using our framework, and assess the resulting flexibility of our processor by sketching several further extensions to the language. Finally, we relate our work to similar efforts and summarize our conclusions.

2 The Object Model: Modules And Operators

With the emergence of complex language mechanisms such as inheritance in O-O programming, it no longer suffices to design language processors based solely on a fixed set of implementation-motivated considerations. Rather, one must secure the design to sound semantic ground. The concepts outlined in this section provide the semantic basis for the design of our metalevel language architecture.

The range of current O-O languages embody varying notions of the *class* concept, each of which differs from others in subtle but important ways. Nevertheless, it is undeniable that the different notions share a common semantic goal: to facilitate the *structuring* and *combination* of software units with well-defined interfaces. We use the term *module* to refer to such software units. Classes traditionally fulfill a variety of roles, including defining modules, defining subtyping relationships, controlling visibility (e.g. via public/protected/private interfaces), constructing instances of a defined module, modifying and reusing existing program units via single inheritance, combining program units using multiple inheritance, resolving name conflicts, etc. This realization motivates the formulation of our central abstraction, *module*, in such a way that it permits aspects of the class construct such as inheritance and visibility control to be "unbundled" as operations generically applicable to modules.

We draw on previous work [BL92, Bra92], which has succeeded in formulating the module notion and operations on modules as a set of *operators* in a module manipulation language called *Jigsaw*. *Jigsaw* is unusually powerful in accommodating differing senses of modules. Bracha and Lindstrom [BL92, Bra92] have given a rigorous formal semantics for *Jigsaw*'s module abstraction, building 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.

In *Jigsaw*, a module is simply a self-referential scope, associating labels (identifiers) with 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. Every module has an associated interface, which comprises the labels and types of all its visible attributes. Types in *Jigsaw* are purely structural, i.e. sets of label-type pairs, without order or type name significance.

Modules may be combined with other modules to achieve the effects of single and multiple inheritance, visibility control, rebindability, and sharability, etc. Such effects are made possible via a suite of module operators (*combinators*) designed to fulfill specific isolatable semantic roles. For concreteness, an example of *Jigsaw* modules and module combination is given in Figure 1, in which a generic surface syntax is used. The modeling power of *Jigsaw*'s module abstraction and module combination operators is fully investigated in [BL92].

3 *Jigsaw*: A Modularity Framework

For expository purposes, *Jigsaw* has been described thus far as a concrete language. The crucial point, however, is that one may view *Jigsaw* merely as an *abstract module manipulation language*. That is to say, it is possible to formulate *Jigsaw* in such a way that it does not prescribe the com-

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 1: *Jigsaw* modules and operators.

putational domain, or the control structures, or even the surface syntax of the concrete language in which it is used. This formulation is facilitated by the use of O-O frameworks, where the concept of *abstract classes* is central. In this section, we present a framework for module manipulation encompassing the module manipulation language semantics presented above. We call this framework the *Jigsaw framework*.

In essence, an O-O framework [JR91] expresses the design of a software system in terms of objects and interactions between them, typically represented using a general purpose O-O 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*[2] instances of classes in the framework. Thus, applications are built by *completing* a framework in specific dimensions delineated by the framework designer. Frameworks mostly comprise abstract classes, which are concretized by an application. As a result, a framework can be thought of as being parameterized on a completion that provides *call back* code — a sort of bi-directional function abstraction. Frameworks thus promote design and code reuse through O-O concepts such as inheritance and polymorphism. Several

[2]Connecting objects constructed from predefined concrete classes [JR91].

frameworks have been developed, first for user interfaces, and subsequently for many other domains as well [Deu89, VL89, WGM88, CIJ+91].

In a reflective language environment such as the CLOS MOP, the framework implementing the metalevel architecture is specified in the language *itself*.[3] However, as mentioned earlier, *Jigsaw* is best viewed not as a concrete language, but as an abstract module manipulation language that can serve as a *framework* for the metalevel architectures of other O-O languages. Hence, we are in a position to distinguish this abstract language from the following two concrete languages: the *framework implementation language* L_f, which is the language used to implement the *Jigsaw* framework, and the *client language* L_c, which is the language for which a processor is to be constructed by extending (completing) the *Jigsaw* framework. In traditional compiler writing, these two languages correspond to the implementation language of a compiler and the language that the compiler is to implement.

3.1 The Abstractions in the Framework

In Figure 2, we present an overview of the abstractions of the *Jigsaw* framework. For the framework implementation language L_f, we adopt a generic O-O language surface syntax that should be fairly easy to understand. Each box stands for an abstraction, with shaded boxes standing for abstract classes (i.e. incompletely specified) while non-shaded ones are concrete (i.e. instantiable). The text within each box is the interface (or protocol) of the corresponding class.

The concrete class `Module` captures the *Jigsaw* notion of module in its broadest conception. Objects of this class represent modules in the client language L_c, each with a set of label-binding pairs initialized via the method `make_module([Attribute])`. Such module objects can be combined with other module objects using module combination operators which are methods in the interface of the class `Module` (cf. Figure 1).

Module objects in L_c are instantiated by invoking the method `instantiate()`.[4] This method returns an object of class `Instance` which represents instances of modules in L_c. The key method that instance objects respond to is `select(Label)`, which when supplied a label, returns its binding. The `select(Label)` method thus corresponds to the notion of sending a message to an instance in L_c, 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 layout for instances and a mechanism by which to use that layout. This facility can be capitalized upon to provide interoperability among different client languages. An ability for *introspection* (examination of meta-information) is provided for `Instance` objects via the method `module_of()`.

For the purposes of typechecking, the *interfaces* of module objects are captured as objects of

[3] It is worth noting that the MOP consists entirely of concrete classes; hence it is more like a completion of a metalevel architectural framework.

[4] *Jigsaw* does not model the notion of object *initializers* (e.g. constructors in C++) explicitly; instead initializers are ordinary methods that are called after instantiation.

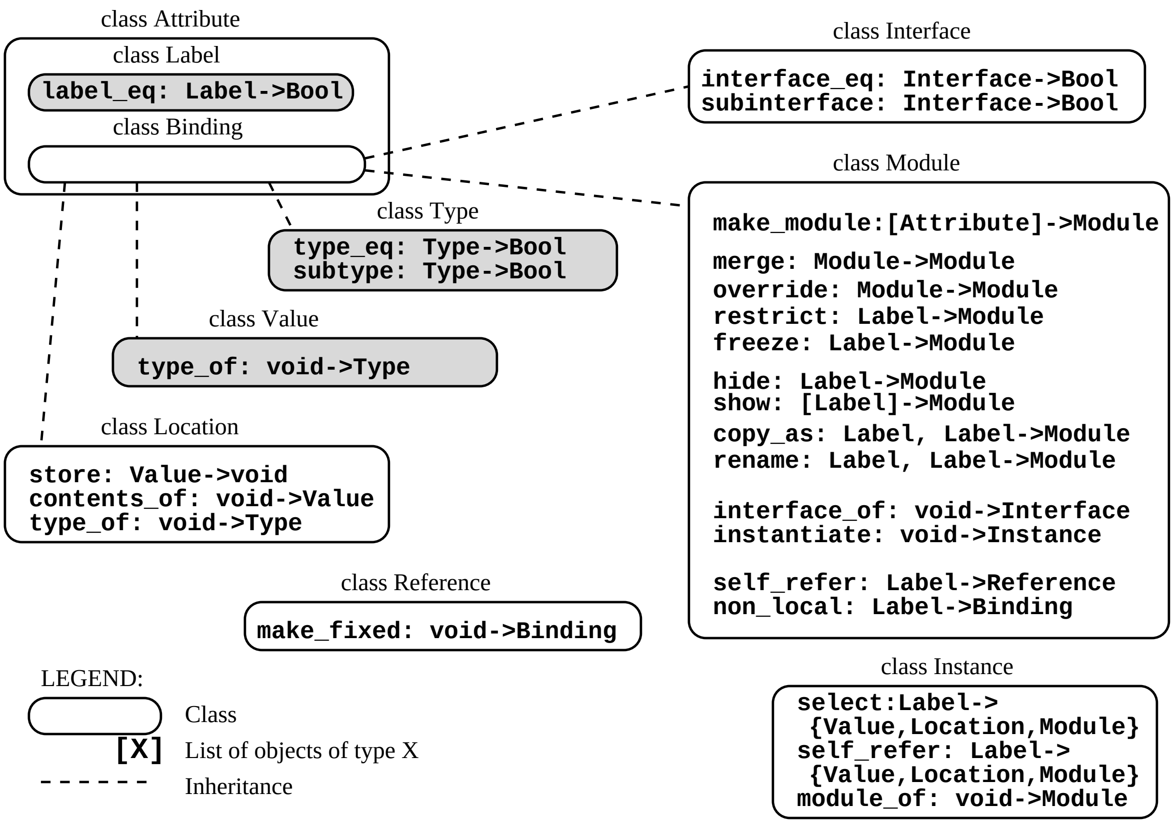

Figure 2: Overview of the *Jigsaw* framework

class `Interface`. When module objects are combined using module operators, their type compatibility is checked by comparing the interface objects corresponding to the modules with the help of methods in class `Interface`. The type checking rules incorporated into the *Jigsaw* framework are explained in detail in Bracha [Bra92].

As explained earlier, the framework provides a rather generic object model (via the module abstraction) and nothing more. As a result, the above abstractions are defined relative to the notions of *value*, *type* and even *label* in a client language, over which *Jigsaw* abstracts. The client language must provide its own concept of values, types and labels. These concepts are therefore incompletely specified abstractions within the *Jigsaw* model, and are specified as abstract classes `Value, Type`, and `Label`. *Jigsaw* requires label objects to supply a notion of label equality via the method `label_eq(Label)`, and value objects to return type objects when queried with `type_of()`. Type objects in turn must supply notions of type equality (`type_eq(Type)`) and subtyping (`subtype(Type)`). A particular client language is implemented by supplying definitions for these methods in these abstract classes, and possibly by extending the functionality (interface) of abstractions, or by adding other abstractions. These definitions and extensions constitute an implementation of the client language.

Mutable state (i.e. instance variables) is modeled in *Jigsaw* via the class `Location`. Location objects hold *storable* values, the exact definition of which is client dependent. The default definition of `Location` comprises value objects and instance objects as storable values, but a particular client language could refine this to include locations (pointers), types, interfaces, or even modules. This exemplifies one virtue of the framework approach to language development — isolation and illumination of the options available to the language designer.

The reader might have noted the correspondence between the above framework abstraction design and denotational models of programming languages [Gor79]. Denotational semantics applies functional programming to abstract over language functionality. Here, we apply a denotational description of modularity in O-O programming to abstract over language modularity. Furthermore, the framework approach is intended to provide the language developer a modular means by which to design *and implement* a language's value domain, type system, etc. relatively independently of each other and independent of abstraction mechanisms in the language. Once the basic elements of the language are designed, the modularity mechanisms available in the *Jigsaw* framework are directly available for incorporation into the language.

A framework is meant to implement *reusable* abstractions. Although the design of the *Jigsaw* framework was motivated by purely semantic concerns, it is currently finding applications in a variety of situations, some of which were unanticipated, e.g. the programmable linker OMOS [OM92]. Indeed, this framework's asserted flexibility benefits are currently being demonstrated in a second generation of OMOS, in which we are incorporating type-safe linkage of object modules. This application attests to the utility of our framework abstractions — although the framework will undoubtedly undergo unforeseen changes as its various completions mature. As has often been observed, repeated reuse enhances and validates the reusability of framework abstractions.

3.2 Encapsulation And Typechecking

Fundamental in the design of the *Jigsaw* module model are the related concerns of encapsulation and type systems. We believe that they are crucial for constructing reliable, readable, and efficient large-scale software. Our notion of encapsulation essentially distills to (i) separating interface from implementation, and (ii) allowing external access to an object only via its interface. The `hide` operator (and its dual `show`) enables encapsulation by removing its attribute parameter from the interface of its module (see Figure 1). The method `select(Label)` of class `Instance` implements access to externally visible attributes of an instance object (i.e. encapsulated access), while self-reference *within* individual objects is accomplished via the method `self_refer(Label)`. *Jigsaw* models *object-level* encapsulation[5], as opposed to class-level encapsulation as found in languages such as CLU [LG86] and C++ [ES90], where objects of a class have access to each other's internals.

A module type system is built into the *Jigsaw* framework. This default type system is *structural* rather than *name-based*, the latter being found in most current O-O languages. The notion of module types is represented by the framework class `Interface`, and module composi-

[5]The strongest form of encapsulation, in which encapsulation walls exist around each individual object.

tion operators verify type compatibility by calling the methods `interface_eq(Interface)` and `subinterface(Interface)` of class `Interface`, which implement module type equivalence and subtyping respectively. These methods in turn rely on the client supplied notions of type equivalence and subtyping defined in the methods `type_eq(Type)` and `subtype(Type)` of class `Type`. Such an architecture makes it fairly straightforward to develop even typeless (i.e. singly-typed) languages since the entire type system relies on the notions of type equivalence and subtyping supplied by class `Type`.

As will be surveyed in Section 4.3, it is possible to design a range of type systems with varying degrees of expressiveness depending upon client language requirements, such as separate compilation and static typechecking.

3.3 Implementation

The generic nature of our module abstraction ensures that it can be represented easily in essentially any existing O-O language. In our prototype implementation of the *Jigsaw* framework, we have chosen C++ [ES90] as our framework implementation language $\mathtt{L}_f$. In this section, we comment on a few aspects of this implementation.

We have included the following supporting classes in our framework: (i) class `Binding`, which is a generalization of all types of entities that can be bound to a label within a module (a value, a location or a nested module to create a label definition, or a type or interface to create a label declaration), and (ii) class `Attribute` which implements a node in a linked list of label-binding pairs, with operations to add, remove, find, etc. such pairs.

Nested modules are modules that are bound to labels within other modules. Attributes within nested modules are permitted to access label definitions in lexically surrounding scopes, i.e. *non-local* attributes. Not unexpectedly, such access is implemented in class `Module` as a private pointer variable `parent` that points to an *instance* of the surrounding module. The functionality of accessing a non-local attribute is encapsulated in the method `non_local(Label)` in class `Module`.

Reference to *self* is an important notion in O-O, and enables dynamic binding, which is typically implemented via a level of indirection using dispatch tables, e.g. *virtual function tables* in C++. A simple form of dispatch is built into the *Jigsaw* framework in the implementation of the methods `select(Label)` and `self-refer(Label)` of class `Instance`. Depending on client requirements, the default can be refined to incorporate alternate dispatch mechanisms [HC92, Cha89].

Interestingly, the *Jigsaw* module model requires a form of delayed binding occurring not at run-time (i.e. dynamically) but rather at module combination time. This is because module attributes are by default rebindable, but can be made non-rebindable at any point by applying the module operator `freeze`. The `freeze` operator makes references to its label argument *fixed*, or static. In our implementation, we capture references to rebindable attributes as objects of class `Reference`, whose method `make_fixed()` makes an attribute non-rebindable. Class `Reference` objects that remain rebindable at instantiation time take part in the creation of a dynamic dispatch table.

Another important implementation issue concerns the provision of methods for *parsing* the surface syntax of client languages. Ideally, we would have an abstract method in class `Module`, such as `parse_module: Stream->Module`, that produces a module object given a *Stream* of characters, as defined by the client. This method would be an *abstract* method since it would construct 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. However, generally available parser technology (e.g. LALR) is based upon monolithic parser specification and does not yet permit *modular* specifications. Therefore, we have chosen to provide a default module surface syntax as a yacc/lex processable grammar file separate from the *Jigsaw* framework itself. The semantic actions of rules in this grammar create and process modules using the framework classes. An alternate idea under consideration is to incorporate modular recursive descent parsers into each of the framework classes.

4 *JigC*: A Module Extension to the Programming Language C

This concludes our general discussion of the *Jigsaw* framework. In this section, we outline the design and implementation of an upwardly compatible module extension of the programming language C [KR88], called *JigC*, which is being developed to showcase and further evaluate the *Jigsaw* module model. In this experiment, we first retrofitted a module system onto an existing language, and implemented a processor for it using our framework. We demonstrate the flexibility built into this language processor by surveying various design extensions to the basic module language, and outlining their incorporation into the language processor. We hope to show that once the more difficult task of retrofitting the module system is accomplished, later extensions can be performed with relative ease.

The objective of this section is to give the reader a flavor of the versatility and (re)usability of our framework, as well as the flexibility and evolvability of the client language processor. We begin by delineating generically the steps in realizing L_c modules starting from the metalevel architecture.

4.1 Realization of Modules

Our approach to characterizing modules involves four levels of abstraction and concretization:

1. [Module abstraction:] This is class `Module`, the framework class representing *Jigsaw*'s notion of modules. Class `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 2.

2. [Module in L_c:] The *Jigsaw* notion of modules tailored to a particular L_c , is defined by providing concrete definitions of these auxiliary classes, and/or by subclassing class `Module` in order to refine or customize it, as appropriate for L_c modules. This realization is introduced by the framework completion, i.e. the implementation of the L_c language processor. For example,

JigC's refined notion of modules presented in Section 4.3 is represented by class `Module_jc`, a subclass of class `Module`.

3. [INDIVIDUAL L_c MODULES:] Once the L_c notion of modules is made complete, individual L_c modules can be created, with specific interfaces, labels and bindings. L_c modules are created as specified in L_c source programs by the L_c processor by creating `Module` objects and invoking the `make_module([Attribute])` method of class `Module`.

4. [L_c MODULE INSTANCES:] Finally, if the concept is supported by L_c, instances (objects, or the compile-time representations thereof) derived from particular L_c module definitions are created as specified in L_c source programs through invocations of the `instantiate()` method of an individual L_c module.

This four-stage process is the way by which the *Jigsaw* module model is exploited in an L_c language processor. It is important for the reader to understand each of the above levels, and to maintain their conceptual separation.

4.2 The Basic *JigC* Module System

A language such as *JigC* that is based on the *Jigsaw* module model is advantageous over similar languages (e.g. C++) for the same reasons that *Jigsaw* modules are advantageous, namely (i) it provides a uniform module model with unbundled inheritance operators, (ii) it supports a static structural (interface-based) module type system, and (iii) it supports nested modules, a powerful feature (e.g. enabling combination of inheritance hierarchies [OH92]). In addition, implementing the *JigC* compiler as a completion of the *Jigsaw* framework brings with it the important benefit of evolvability, which is the subject of Section 4.3.

The interactions of the module system with the client language (C, in our case) are numerous, and sometimes subtle. For example, a requirement in the design of *JigC* is backward compatibility with C, which implies that existing C programs can also be viewed as *JigC* program. From the *JigC* point of view, a program *file* may itself be regarded as a module in which C declarations[6] (constant, variable and function declarations), C function definitions, and *JigC* declarations/definitions are module *attributes*. A complete *JigC* program is thus a fully concrete module that contains a function-valued attribute called *main*. It might not be immediately obvious what it means to instantiate an ordinary C program file when viewed as a module — in *JigC*, each *execution* of the program is viewed as an instantiation. Furthermore, since location attributes are by default regarded as *per-instance* locations in the framework, each instantiation of a module containing such an attribute will get a new copy of the location — consistent with ordinary models of program execution.

[6]The term *declaration* used in the context of C has a different meaning than that of *Jigsaw*. In C, a variable declaration allocates storage, while in *Jigsaw*, a declaration specifies a type. In this paper, the term declaration is used in the *Jigsaw* sense unless noted otherwise.

Existing C Syntax	*JigC* Attribute Semantics
`int x;`	Location (of type `int`) definition
`extern int x;`	Location (of type `int`) *declaration*
`int x = 0;`	Location (of type `int`) definition w/initialization
`static int x;`	Location (of type `int`) definition subjected to `hide`
`const int x = 0;`	Value (of type `int`) definition
`extern const int x;`	Value (of type `int`) declaration
`int foo (float x) { ... }`	(function) Value (of type `float->int`) definition
`extern struct s { ... }`	(aggregate) Value (of type `s`) declaration

Figure 3: *JigC* attribute syntax and semantics examples

```
const int g = 3;
module Point : PointType {
  int x = 0;
  Boolean atOrigin () { if (x == 0) return TRUE; else return FALSE; }
};
main () { Point p; p.x = g; ... }
```

Figure 4: A simple *JigC* program

If existing ordinary C programs are to be viewed as *JigC* modules, each top-level C declaration or definition must correspond to a *JigC* module attribute. Figure 3 shows such correspondences. In addition, *JigC* introduces new keywords related to the module system such as `module`, `merge`, `override`, etc. Figure 4 shows a simple *JigC* program in which the outermost module (i.e. the entire program file, which we will refer to as *FILE*) contains three attributes: a value definition `g` of type `int`, a module definition `Point` of type `PointType` (which in turn contains two attributes: a per-instance location definition `x` of type `int` with initial value `0`, and a (function) value definition `atOrigin` of type `void->Boolean`), and a (function) value definition `main` of type `void->void`. The modules *FILE* and `Point` are both concrete, i.e. all their attributes are defined and not just declared, and hence they are instantiable.

In implementing a processor for *JigC* as a completion of the *Jigsaw* framework, we must first understand the notion of identifiers in our client language C. We implement this notion as a subclass of the framework class `Label`.

Next, we independently describe the value (and corresponding type) domain of C — primitive data types such as integers, floats, characters, etc.; aggregate types such as structs, and unions; and function types. The framework class `Value` is subclassed into several classes to implement each of the above, and the class `Type` is subclassed to implement corresponding notions of type. Since C relies on name-based typing, type equivalence is name equivalence in most cases; the exceptions are type equivalences defined in the language (e.g. `int` is equivalent to `short int`) and types introduced via `typedefs`. The subtyping relationship surfaces in two situations: (i) type conversion rules defined in the existing language (e.g. C's integral promotion: `char` to `int`), and (ii) a *JigC* augmentation of C's function typing: the contravariant subtyping rule for functions, which is necessitated by the existence of a subtyping relation on `Instance` types. The function `subtype` of class `Type` is implemented by taking the above rules into account and also the relations of reflexivity and transitivity.

In the basic *JigC* language, class `Location` is subclassed in order to incorporate location objects as storable values (for modeling C pointers). The default semantics of the framework class `Location` includes only value objects and instance objects as storable values.

The above specializations of the abstract framework classes together with the default semantics provided by the *Jigsaw* framework accounts for a large portion of our language implementation. The rest of the language, primarily control structures, can be dealt with using traditional techniques. We have chosen to translate *JigC* programs to C programs (as a result, our translator is essentially a pre-processor to the C compiler). Although we did not have to specialize the semantic notion of modules, we do find the need to refine the functionality of modules in two areas due to compilation requirements. First, the notion of instantiation is refined in order to perform layout and initialization for runtime instances, and correspondingly, the methods of class `Instance` are refined. Second, methods named `translate_to_C()` are added to each of the framework abstractions.

The final task is to reverse-engineer the existing C language grammar to incorporate the module system, and add semantic actions that utilize the framework classes and their refinements. The architecture of our *JigC* implementation is shown in Figure 5.

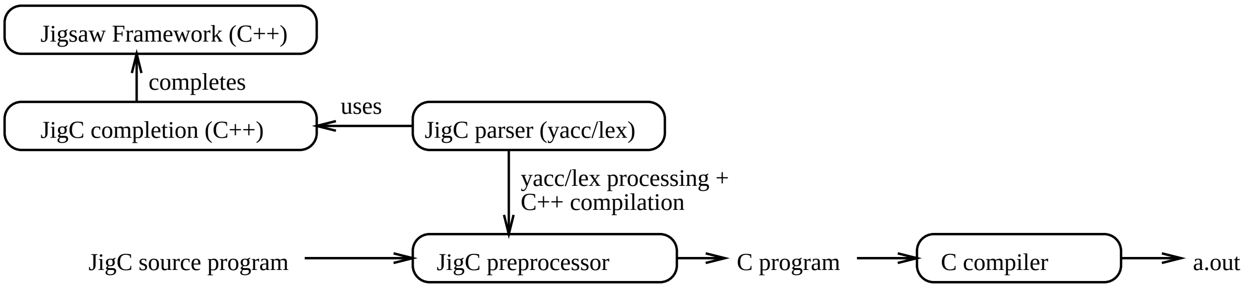

Figure 5: *JigC* implementation architecture

4.3 Extensions for Advanced Modularity Features

The central purpose in our developing a module extension to a legacy language was to experiment with advanced modularity concepts. Therefore, it is extremely important to us that the design *and implementation* of our experimentation vehicle be highly extensible. Our framework approach to language implementation has enabled us to isolate the parts of our language processor that need to change as the design of the language evolves, and specify these parts in an evolvable manner. To illustrate, consider the incorporation of recent advances in static type checking, involving programmer access to *self* and its type.

By default, *Jigsaw* models implicit access to *self*, i.e. module attributes refer to siblings by label. A more expressive static type system would demand explicit programmer access to self and its type. Incorporation of explicit access to self is quite straightforward, but type checking the type of self is more problematic.

Using our framework, the type of self, which is an *instance* type[7], can be designed as a distinguished object of class `Interface`. Furthermore, if function parameters of self type are to be permitted in a client, as described extensively in recent literature [Bru93, Bru92], two major refinements to the default type system would be required. Firstly, in the default formulation of *Jigsaw*, the `hide` operator permits removing an attribute from a module's public interface *after the fact*, i.e. module interfaces can evolve non-monotonically. As a result, usages of incoming function parameters that are declared to be of the type of self cannot be type checked at module definition time, since the type of self could arbitrarily change under inheritance and as a result, no assumptions can be made regarding the type of self. In order to alleviate this situation, class `Attribute` must be subclassed to model public attributes, i.e. methods that cannot be hidden. Secondly, there would be a need for type checking *inherited types*[8] — this can be done by subclassing class `Interface` and class `Type`. Language design issues such as these are discussed in detail in the literature [HC90, Bru93, Bra93] — the key point is that a wide range of type systems can be accommodated as refinements of the basic formulation of the *Jigsaw* framework.

In its current form, a function defined within a *JigC* module has access to the instance that it is executing within via the keyword `self`. In addition, the type of `self` can be denoted using the keyword `selftype`[9]. As described above, attributes that cannot be hidden are specified by prefixing them with the keyword `public`. A static type error results if an attempt to `hide` a `public` attribute is made. An example program with these typing features is shown in Figure 6.

In the rest of this section, we enumerate several other design extensions to the basic *JigC* module system, and outline their flexible incorporation into the language processor. Some of the following extensions have been incorporated into *JigC*, while others are still under development.

- *Per-module shared locations.* Consider the incorporation of per-module locations, i.e. locations that are shared by all instances of the containing module (as in `static` members in C++). This requires (i) syntactically, incorporation of a new keyword (such as `shared`) into the grammar, (ii) class `Location` to be subclassed to class `sharedLocation`, and (iii) refinement of the `instantiate` method of class `Module` to share objects of class `sharedLocation`.

- *Name-based typing.* If it is necessary to design a name-based type system for a particular client language, this can be achieved by incorporating the concept of brands [Nel91, BG93] into a subclass `brandedInterface` of class `Interface`, and refining the corresponding constructor and equivalence methods.

[7]Although instance types are generally regarded as distinct from module types (i.e. interfaces), they can be modeled as `Interface` objects.

[8]Types that are not subtypes but share a similar recursive nature[CCHO89].

[9]This corresponds to the bound variable *MyType* in [Bru92]

- *Bundled inheritance semantics.* If a client requires retention of particular compound inheritance semantics, such features can be modeled using *Jigsaw* operators. For example, one might desire to reconstruct an existing language such as C++ using the *Jigsaw* approach so that future evolution, such as an enhanced type system or inheritance semantics, is possible.

- *A language for linking.* As mentioned in Section 3.1, *Jigsaw* has another embodiment in OMOS, which offers programmable combination of compiler-emitted object files. In addition to supporting the module combinators illustrated in Figure 1, OMOS extends the *Jigsaw* framework in several respects, including (i) address space mapping constraints, and (ii) constructed module caching, reuse and sharing. The result is a greatly enlarged conception of object file (module) manipulation and management, cast as a pervasive system service, under which many *value added* features are deliverable, including portable shared libraries [OBLM93], function interposition [BCLO93], and dynamic program monitoring and reorganization [OMHL93].

- *Concurrent O-O programming.* It is increasingly evident that the relationships between sequential and concurrent conceptions of O-O programming are inadequately understood. In particular, several researchers have reported that inheritance, as commonly understood for sequential O-O languages, gives rise to semantic anomalies and violations of encapsulation when applied to concurrent O-O languages. Various remedies have been proposed (e.g. [L9̈3, Mes93]), but none directly addresses the issue of migrating existing sequential O-O code to concurrent settings. We conjecture that the *Jigsaw* framework, enhanced to support asynchronous message dispatch via *synchronization condition predicates* managed as rebindable module attributes, will provide a more general and satisfying solution. We plan to test this conjecture by porting *JigC* to the pseudo concurrent environment provided by Cthreads [CD88].

- *Nested Modules.* Thorough incorporation of nested modules in a language opens up new semantic avenues, e.g. sharing, and several software engineering possibilities, e.g. combination of inheritance hierarchies. While the *Jigsaw* module model provides the capability for defining and using nested modules, this has not yet been taken advantage of in the current *JigC* language, and is being investigated in a continuing effort.

- *Persistence.* *Jigsaw* arose within the context of the *Mach Shared Objects* project, which is building an persistent store for C++ and CLOS objects. Persistence raises many new requirements, including (i) metalevel object information (e.g. class objects, or "dossiers"), and (ii) resolution of semantic issues such as whether persistence should be accorded to `shared` module attributes. The *Jigsaw* framework directly provides a basis for dealing with issue (i), in that modules (classes) already exist as tangible objects in our metalevel architecture. One option in dealing with issue (ii) is to regard such an attribute as a *persistent* attribute, since each execution of the program file is regarded as an instantiation of the *FILE* module. This

```
module Point : PointType {
 public:
  extern int x;
  selftype moveOne () { self.x = self.x + 1; return self; };
};
module NewPoint : NewPointType =
  Point override {
   public:
    int x = 0;
    Boolean eq (selftype p) { return (self.x == p.x); };
  };
main () {
  NewPoint p1, p2;
  p1.x = 13;
  printf ("%d", p1.moveOne().x);
  while (!p1.eq(p2))  ...
  ...
}
```

Figure 6: An extended *JigC* program using `selftype`

involves the creation of a subclass of class `sharedLocation`, say class `persistentLocation`, and add corresponding persistence semantics to it and the `instantiate()` method.

We are encouraged by our initial findings about the utility of our metalevel language architecture, and we envisage future work in the directions indicated above. We also intend to evolve *JigC* into genuinely useful language (hence a competitor to related languages like C++), believing that the principles upon which it is based (e.g. structural typing with brands) are more advantageous. We also foresee the exploration of fundamental issues concerning encapsulation, polymorphism and type checking in O-O programming using this framework.

5 *Jigsaw* and Reflective Systems

The *Jigsaw* framework approach to building language processors has a relation to *reflective systems*, and is somewhat similar to languages with meta-object facilities such as the CLOS MetaObject Protocol (MOP), and Smalltalk-80 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.

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 from the requirements of diverse language designs already in existence. As already mentioned, the *Jigsaw* framework was specifically designed to facilitate the construction of modular language processors and systems, and finds applications in the interoperability among languages, linkers and libraries. However, the *Jigsaw* framework can be used for many purposes that the CLOS MOP has been put to use, notably persistent objects [Pae88, Lee92].

6 Conclusions

A framework-based approach to language processor design and implementation has been described. This approach, called *Jigsaw*, relies on an abstract conception of software modules, refineable in several dimensions to characterize a large space of specific module formulations. A central property of this framework is its exploitation of this module conception on two levels: within its own organization, and within the language systems definable through it. Like traditional denotational semantics, which uses functional programming to describe language functionality, *Jigsaw* uses modular programming to describe language modularity. Because frameworks are conveniently realizable in today's O-O languages, the *Jigsaw* approach directly lends itself to experimentation. We have constructed a prototype of the *Jigsaw* framework in C++, which we have extended to *JigC*, an extensible compiler for C-based O-O programming languages. Our initial experience has encouraged us to apply the *Jigsaw* approach to more diverse aspects of module combination and management, including programmable linkers, persistent object stores, and concurrent object systems.

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 the paper. The insights and support of Charles Clark, Douglas B. Orr, and all other Mach Shared Objects project participants are also gratefully acknowledged.

References

[AW82] E. A. Ashcroft and W. W. Wadge. Prescription for semantics. *ACM Transactions on Programming Languages and Systems*, 4(2), April, 1982.

[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. Module management as a system service. Unpublished paper, July 1993.

[BG93] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proc. OOPSLA Conference*, Washington D.C., September 1993. ACM.

[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.

[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.

[CCHO89] P. Canning, W. Cook, W. Hill, and W. Olthoff. Interfaces for strongly-typed object-oriented programming. In Norman Meyrowitz, editor, *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 457–467, 1989.

[CD88] Eric C. Cooper and Richard P. Draves. C threads. Draft report, Mach Project, Carnegie-Mellon Univ., 6 March 1988.

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

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

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

[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.

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

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

[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.

[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.

[HC92] Shih-Kun Huang and Deng-Jyi Chen. Efficient algorithms for method dispatch in object-oriented programming systems. *Journal of Object-Oriented Programming*, September 1992.

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

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

[JS80] N. D. Jones and D. A. Schmidt. Compiler generation from denotational semantics. In N. D. Jones, editor, *Semantics-Directed Compiler Generation*, pages 70–93. Springer-Verlag, Berlin, 1980. Lecture Notes In Computer Science Number 94.

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

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[L̈93] Klaus-Peter Löhr. Concurrency annotations for reusable software. *Communications of the ACM*, 36(9):81–89, September 1993.

[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.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* The MIT Press, Cambridge, MA, 1986.

[Mes93] José Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In *Proceedings ECOOP '93*, LNCS, Kauserlautern, Germany, July 1993. Springer-Verlag.

[Nel91] Ed. Greg Nelson. *Systems Programming with Modula-3.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[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.

[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.

[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.

Last revised October 21, 1993