The Design of Object-Oriented Meta-Architectures For Programming Languages*

Guruduth Banavar and Gary Lindstrom
Department of Computer Science
University of Utah, Salt Lake City

Abstract. This paper is a survey of the design of four object-oriented meta-level architectures for programming languages. We present overviews and compare the salient features of the meta-architectures of Smalltalk, Common Lisp Object System (CLOS), a Scheme Compiler, and Etyma, our framework for modular systems. This comparison clarifies important architectural aspects of the surveyed systems, such as the space of concepts captured by the architectures, and the abstractions that embody similar language concepts across the architectures. We find that there are considerable differences in the goals and conceptions of these architectures, yet they can all be used for similar applications. Finally, we point out some strengths and weaknesses of the architectures surveyed.

1 Introduction

Object-orientation is a popular design technique that has been used to model application domains of all varieties [1]. A recently emerging trend is to apply the object-oriented (O-O) method to the design of O-O language processors themselves, thereby harnessing the much touted advantages of abstraction and reuse in this domain also. For example, the Meta Objects of the CLOS MetaObject Protocol (MOP) [2] is an O-O model of certain useful concepts in CLOS. A recently proposed O-O model for a compiler for a non O-O language, Scheme, [3] is another example. One of the earliest O-O programming systems, Smalltalk-80 [4], was itself built upon an intricately interconnected group of meta-classes.

The above languages embody O-O meta-level architectures, or meta-architectures for short, in the sense that they model the fundamental concepts in the language, such as *class* and *method*, as interacting meta-classes. This has resulted in reflective, flexible, and extensible language designs. Many of these advantages stem from the fact that reified meta-classes are candidates for systematic reflective access. That is, a system that has a well-designed meta-architecture can essentially provide users not only with its standard

*This research was sponsored by the Advanced Research Projects Agency (DOD), monitored by the Department of the Navy, Office of the Chief of Naval Research, under Grant number N00014-91-J-4046. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Advanced Research Projects Agency or the US Government. Contact author: G. Banavar, Computer Science - 3190 MEB, University of Utah, Salt Lake City, UT 84112 USA, e-mail *banavar@cs.utah.edu*, phone +1-801-581-8378, fax +1-801-581-5843.

interface, but with an alternative interface — a "side door" to the internal architecture, which is typically a subset of the meta-architecture interface. Information access and refinement via this alternative interface can enable applications to fine-tune a language implementation to suit its particular needs. Meta-classes can be specialized to suit specific tasks using standard O-O techniques such as inheritance. In a compilation setting, meta-classes can even be specialized to statically optimize run-time data layout or generate optimized code for particular special cases.

It is important to clarify the relationship between the concepts of *meta-architecture, reflection*, and *metaobject protocol* (MOP). A meta-architecture models, systematically implements, and documents the fundamental concepts of a system. A meta-architecture is O-O if the concepts are modeled as collaborating classes. A system is reflective if its users have introspective (i.e. read) and/or intercessory (i.e. modification) access to the internal implementation of the system. Finally, a MOP documents and illustrates a disciplined method of reflective access to a carefully chosen subset of a system's O-O meta-architecture.

The semantics of a programming language is rarely made formally explicit in the language implementation, let alone made usefully accessible from within the language itself. This may be due to the fact that the design of programming languages is generally considered to be an amorphous creative activity, carried out by the best experts in the field. However, this need not necessarily be the case — an important point of this paper is that the design discipline encouraged by object-oriented methods can be fruitfully applied to the design of programming languages themselves. Furthermore, a significant part of the semantics of a language can be reified as an O-O meta-architecture. A well-designed meta-architecture enables reuse, reflection, and the design of a suitable MOP, and thus brings the advantages mentioned above.

The traditional architecture for processing high level languages involves the following, usually separate, languages: the source language, the target language (this is not significant in the case of direct interpreters), and the processor description (implementation) language. An important observation is that the meta-architecture of a language is expressed in its processor description language. From this it follows that a language can have multiple meta-architectures corresponding to multiple processor descriptions, each designed with different requirements. It also follows that a language's meta-architecture need not necessarily be meta-circular, i.e. expressed in the source language itself. In fact, in order to have an O-O meta-architecture, a language need not even be object-oriented. However, its implementation language must. The meta-architectures of dynamic languages such as Smalltalk are meta-circular since a large part of the language is implemented using the language itself, as is its meta-architecture.

In this paper, we contrast the three meta-architectures mentioned above, Smalltalk, CLOS MOP, and the Scheme compiler MOP, along side our own meta-architecture based on a language called Jigsaw [5]. Smalltalk and CLOS are general-purpose O-O languages that enjoy significant followings. Smalltalk was built ground-up based on a remarkably coherent meta-architecture, while the meta-architecture for CLOS was retrofitted onto the language. The Scheme compiler MOP shares the goals of the CLOS MOP, but is significantly different from the above two since Scheme is not O-O, and its MOP is compile-time oriented. Finally, our framework, Etyma, attempts to generalize and bring together many of the concepts from the above meta-architectures.

The paper continues by discussing the design issues investigated in this survey, followed by detailed descriptions of the four meta-architectures under consideration. Finally, we provide a summary of architectures and conclude.

2 Design Issues

The design of meta-architectures for languages is driven by various considerations. In this section, we outline some of the issues that govern the design of meta-architectures, the main categories being (i) the pre-stated design goals of the language as governed by the requirements of applications, and (ii) the requirements of semantic models of languages, also driven by applications — i.e. how the abstractions of meta-architectures must capture the crucial semantics of their languages.

Goals and Application Requirements

Consider the competing goals of *generality vs. backward compatibility.* A primary requirement in the design of the CLOS MOP was backward compatibility with several existing LISP Object systems which were pairwise incompatible. The refineability of the object model enabled by the MOP essentially brought about the backward compatibility. Hence, it was sufficient to model just the object system in the meta-architecture, in such a manner that backward compatibility can be achieved. One can imagine that the meta-architecture in such a scenario could be markedly constrained by the existing object models. On the other hand, the Smalltalk and Etyma meta-architectures were built from scratch based upon a uniform model, i.e. every concept in these systems is modeled in the meta-architecture. Moreover, Etyma was designed from the start with the explicit purpose of abstracting semantic commonalities in module-based languages and systems. As a result, the abstractions provide a clean module and inheritance model, leaving the rest of the language design open.

An important application of meta-architectures is the support for flexibility and extensibility of a language via *reflection and MOPs.* In addition to introspective access, a MOP typically provides intercessory access to meta-objects, making it possible to refine them incrementally using standard O-O techniques and hence amend the existing language design itself. The user can define new meta-object classes as specializations (subclasses) of the standard meta-object classes, refining their behavior as necessary. The most important goal of the CLOS and Scheme compiler meta-architectures is to provide a metaobject protocol for users. This has led to the pragmatic design goals of (i) controlled and carefully documented user extensibility, (ii) interoperability of separately designed extensions, (iii) efficiency via user specialization, and (iv) ease of use. In the context of MOP design, there is a tradeoff between implementor freedom and user extensibility. The CLOS MOP designers deal with this tradeoff by explicitly specifying restrictions on the usage of the MOP.

Given reflective access to its meta-architecture, a language's source programs are made up of base language code interspersed with meta-code, i.e. the code that accesses the meta-architecture. One design issue in such systems is the *execution time of meta-code.* Dynamic environments like Smalltalk and CLOS require meta-code to execute at run-time, while the Scheme MOP runs meta-code at compile-time. Dynamic architectures exhibit meta-circularity, and hence a tight coupling between the language and the meta-architecture. While this coupling enhances application development flexibility, it causes the meta-level architecture to become difficult to disentangle from their base languages for separate reuse.

The need for self-applicability (*meta-circularity*) is an important design issue. It is a fundamental requirement in the dynamic environments of Smalltalk and CLOS MOP. It is not even possible in Scheme since it is a non O-O language with an O-O meta-architecture. The requirements of static typing and separate compilation make it impossible to express the Jigsaw language (on which Etyma is based) using itself. The details of this are beyond the scope of this paper [6].

A language meta-architecture supports *reuse* of design and code just as a domain specific O-O framework does. O-O frameworks for several domains have been constructed [1]. Similarly, an O-O framework can be used as a reusable domain model for O-O languages and systems — indeed, this is a primary goal of Etyma.

An important use of meta-architectures is as an *aid in understanding/maintaining* the system. Another use is the *construction of program analysis tools* such as browsers and debuggers. Specific meta-architectures have also been used for other applications such as *persistence*.

Requirements of Semantic Models

In addition to general goals such as the above, meta-architecture designs have several semantic requirements. For instance, type-checking is an important issue that must be taken into early consideration when building a meta-architecture. Although meta-architectures are considered extensible, the design decisions in the area of typing built into existing architectures pervade the entire model, and make it hard to retrofit significant static semantics. Another example of a fundamental requirement is the semantics of inheritance.

Feature	Smalltalk-80	CLOS MOP	Etyma
Inheritance	Single	Multiple	Unbundled
Encapsulation	Object-level	(none)	Object-level
Method dispatch	Single	Generic/Multi	Single
Static Typing	No	No	Structural

Figure 1: Selected O-O semantics of surveyed languages

By its very nature, the meta-architecture of a language captures and constrains the semantics of the base language. The space of high-level language semantics is broad and a subspace of it must necessarily be carved out by a particular meta-architecture. The larger the subspace, the more complex and potentially less useful the meta-architecture. Once the subspace is chosen, the particular way in which semantic concepts within this subspace are modeled is also significant. Furthermore, the location of the point representing the base semantics of the language must be chosen within this subspace. Figure 1 tabularizes a sample of O-O semantic choice points of the languages surveyed here. In the following sections, we describe the specific semantics captured by meta-architectures for O-O languages in detail, including support for inheritance, encapsulation, method dispatch, instantiation, static typing, and abstract classes.

We now turn to a more detailed treatment of specific meta-architectures.

3 Smalltalk

Smalltalk is based on a uniform model of communicating objects. It has a small number of concepts — object, class, instance, message, and method. Every concept in the system is modeled as an object, either instantiable (class object) or not (instance object). The most primitive low-level operations in the system are delegated to a virtual machine. Objects communicate via messages; the semantics of messages are implemented by receivers as methods.

Smalltalk's notion of objects is captured by class `Object` which provides the basic semantics, including message handling, of all objects in the system. The semantics of classes is captured by class `Class` along with its superclass `Behavior` which defines the state required by classes, such as for instance variables and a method dictionary. Further, the class `CompiledMethod` embodies the notion of a class' method; this class defines a method `valueWithReceiver:` to evaluate itself. The O-O semantics of Smalltalk captured by the meta-architecture is given in Figure 2.

Inheritance	The class `Class` implements a message `subclass:...` which accepts one class parameter, the superclass, thus implementing single inheritance. The subclasses of `Class`, which are the meta-classes of individual classes, inherit the method `subclass:...`, thus individual class objects also respond to this message. The assumption that classes have a single superclass permeates the system. Inheritance of state and methods is captured by the superclass of `Class`, class `Behavior`, which implements methods to compute the set of instance variables and methods available to instances of a class.
Message handling	Class `Object` defines a method `perform:withArguments:` that handles message dispatch using a primitive method of the Smalltalk virtual machine. There is also a class `MessageSend` that captures the notion of a message-send. However, for efficiency, an instance of this class is not created for every message-send in the system.
Encapsulation	Instance variables are encapsulated in Smalltalk. Method handling code searches only the method dictionary of a class, but not the instance variables. Method objects have access to instance variables since they refer to a scope object that records the variable objects accessible within that scope.
Instance creation	Class `Behavior`, the superclass of class `Class`, defines a message `new:`, which calls a primitive message to create an instance of the receiver (which must be a class).

Figure 2: Smalltalk's O-O semantics

Smalltalk is a "dual hierarchy" language, as are most object-oriented languages. That is, it has a cleanly articulated class-subclass hierarchy as well as a class-instance hierarchy. In most languages, however, the class-instance hierarchy is not interesting since it comprises only two levels — that of all classes and all instances. In Smalltalk, this hierarchy is deeper, and is recursive, as described below.

Every object in Smalltalk is an instance of some class. Since classes themselves are objects, each class object is an instance of yet another class, usually referred to as a *metaclass* object. For example, a class `Foo` is an instance of its metaclass, given by the expression `Foo class`. Such metaclass objects are themselves instances of an ordinary class called `Metaclass`. The metaclass of class `Metaclass` itself is given by `Metaclass class`, which is also an instance of class `Metaclass`, just as `Foo class` is. The above recursion puts an end to the infinite regression of metaclasses.

Consider the class-subclass hierarchy of metaclasses. Every class in Smalltalk inherits from class `Object`; hence the subclass hierarchy is a singly rooted tree. The class `Metaclass` mentioned above is also a subclass of `Object`. The instances of class `Metaclass`, such as `Foo class`, `Metaclass class`, and even `Object class`, are all (meta)classes. These metaclasses are subclasses of class `Class`, which is a subclass of class `Object`[1].

In Smalltalk, the meta-architecture is really "infinitely open," in the sense that every single concept in the system (except some primitive operations that are performed directly by the virtual machine) is captured as an object which users can not only specialize, but also browse and access, i.e. directly edit and modify, which of course is strongly discouraged.

4 CLOS MOP

As mentioned earlier, a primary requirement in the design of CLOS was backward compatibility with existing LISP systems. It was recognized that these incompatibilities could be reconciled if a family of languages, rather than a single one, were defined. Thus, the CLOS meta-architecture was designed to facilitate modeling an entire *space* of language designs, with the default CLOS design being a distinguished point. Furthermore, a protocol (MOP) has been carefully designed and documented to access this meta-architecture usefully.

The CLOS object system supports the standard concept of *classes*, which can be instantiated into *instances* [7]. Class attributes are called *slots*. A distinguishing feature of the CLOS model is the notion of *generic functions* which are defined independent of any class, and can be specialized into *methods* that are applicable to specific classes. Generic functions can be dispatched based on *multiple* arguments (multi-methods).

The CLOS meta-architecture specifies the following basic meta-object classes corresponding to the basic concepts of the language: `class, slot-definition, generic-function`, and `method`. All user-defined metaobjects must be designed to be subclasses of one of the above meta-object classes. The specified default semantics of the CLOS language are embodied by specializations of the above classes; with names beginning with `standard-..`, e.g. `standard-class`, and `standard-method`. The manner in which the meta-architecture captures basic O-O semantics in CLOS is given in Figure 3.

The class-subclass hierarchy of the CLOS meta-architecture is as follows. At the root is class `t` which has one subclass `standard-object` capturing the semantics of all objects in the system. Every class created in the system must have `standard-object` as its superclass. One subclass of `standard-object` is the class `metaobject`, of which the basic meta-object classes mentioned above are subclasses.

The class-instance hierarchy of CLOS essentially has four levels. Individual CLOS classes are instances of class `class` or one of its subclasses. Class `class` is an instance of (its own subclass) `standard-class`, as are most other meta-object classes.

The CLOS MOP has functions for systematic *introspective access* to its meta-objects. For example, the programmer can access the class meta-object of a given object, the

[1] The actual subclass hierarchy of Smalltalk is slightly more involved than what is described here, due to the desirability of symmetric class and metaclass hierarchies, but the given description will suffice for this discussion.

Inheritance	Generic functions specialized on the `class` metaobject class implement the semantics of multiple inheritance. A *class precedence list*, i.e. a total ordering on a class' superclasses, is computed by the generic `compute-class-precedence-list`. The generic function `compute-slots` computes the full set of slots accessible from instances of the class. The semantics of slot property union is implemented by `compute-effective-slot-definition` which is called by `compute-slots`. The generic function `class-default-initargs` computes the full set of initialization arguments required by the class.
Generic invocation	A *discriminating function* associated with a `generic-function` metaobject provides the semantics of (multi-)method dispatch. The discriminating function is computed by a generic function `compute--discriminating-function`. Dispatch proceeds by first finding the set of applicable methods for the given set of arguments from the set of all methods associated with the generic function metaobject, via `compute-applicable-methods`, and computing an *effective method* via `compute-effective-method`.
Slot access	The function `slot-value`, a wrapper for the generic function `slot-value-using-class`, is used for slot access. The generic function itself is specialized to class and slot metaobjects implementing the semantics of slot access in CLOS objects.
Instance creation	The generic function `make-instance` and `allocate-instance`, both specialized to `class` metaobject class, implement instance creation. Prior to creating an instance of a class, it is *finalized* by computing the actual structure of the class as described under "inheritance" above.

Figure 3: CLOS's O-O Semantics

class meta-object's name, superclasses, slots (each of which is a meta-object on its own), subclasses and methods. The details of each slot meta-object, generic function, and method can also be accessed. Using these functions, it is possible to, for example, reconstruct a textual description of an object's class.

CLOS MOP is a *layered* protocol, i.e. the protocol specifies meta-architecture functionality at various levels of detail, with higher levels delegating work to lower levels, so that user-refinement can be made at various granularities of semantics. For instance, a top layer protocol concerned with inheritance is the generic function `finalize-inheritance`, which delegates to the next layer, `compute-class-precedence-list` and `compute-slots`, which further delegates to `compute-effective-slot-definition`.

A large number of applications that the CLOS MOP can be put to are illustrated in [2]. These include specialized classes such as counted classes and encapsulated classes. CLOS MOP has also been utilized to provide a significant persistence facility [8].

5 Etyma

Etyma is a general meta-level architecture for O-O languages realized as a C++ framework (in the sense of [1]). The primary abstractions of Etyma are based on a language called Jigsaw, a module manipulation language designed to model the semantic foundations of object-orientation, especially *inheritance*, in all its forms. A basic premise of this work is that O-O concepts, properly formulated, can be applied not only to traditional programming language design, but for the broader design and implementation of O-O programming systems, such as linkers/loaders, library management tools, configuration management systems, type checkers, etc.. We name our framework Etyma (the plural of "etymon," taken from the etymology of "etymology") since it is a collection of root concepts from which other concepts are formed by composition or derivation.

In Etyma, as in Jigsaw, the central concept is that of a *module*, akin to a *class*, which can be informally defined here as any software unit that provides a set of services as specified by its interface. A module consists of a set of labels (identifiers) each associated with either (i) a *value* (e.g. an integer, a function) in a language's value domain, or a *type* in the language's type domain, or (ii) a *location*, in which case the label corresponds to a mutable instance variable, or (iii) a (nested) module or its *interface*. The key characteristic of this model is that modules can be combined using a suite of unbundled and general module *combinators* to achieve various effects of inheritance, sharing, and encapsulation. A summary of the key semantics captured by the meta-architecture is given in Figure 4.

Inheritance	The semantics of the usual kinds of inheritance is supported by some combination of the primitive operators `merge`, `override`, and `copy_as`, with various other effects achieved via the operators `rename`, `restrict`, and `freeze`. All of these operators are implemented as methods of class `Module`.
Typing	A static type system with subtypes and inherited types is supported. The structural type of modules is captured by class `Interface`. Etyma also has a hierarchy of type meta-classes to capture the type space of programming languages.
Encapsulation	Supported by the `hide` operator of `Module`. Hidden attributes are removed from a module's interface, and are accessible only by a class' own methods.
Method dispatch	Supported by the `select` method of class `Instance`. In the default case, `select` dynamically dispatches on attribute name.
Abstract classes	Modules can have attributes whose types are declared but which are not defined. Such modules correspond to abstract classes, and cannot be instantiated.
Instantiation	Supported by the method `instantiate` of class `Module`, which returns an object of class `Instance`.

Figure 4: Etyma/Jigsaw O-O semantics

The class-subclass hierarchy of Etyma has class `Etymon` at the root. A subclass `TypedValue` embodies the typed value domain of languages, and another subclass `Type` embodies the corresponding type domain. Class `Module` is a subclass of `TypedValue`, as is class `Instance`, but via class `Record`. There is a parallel type hierarchy with classes `Interface`, `InstanceType`, and `RecordType`.

In [9], we have described a preliminary C++ prototype implementation of Etyma. The design of abstractions in Etyma has been guided mostly by semantic concerns, with ideas based on a denotational description of the Jigsaw language. Etyma can be used to describe and build processors for many systems that can be construed to be module-based. Languages that are derived from the framework are called *client* languages, and processors for them are constructed by *extending* the framework. The client language is in general unrelated to the framework implementation language — an extension of Modula-3, Modula-π, is presented in [5], and examples of simple languages based on C++ are given in [9]. Etyma is being used as the meta-architectural framework for a larger initiative for evolutionary support for modular architectures, in which a module-based server-style linker/loader is being designed as an extension. In this extension, UNIX ".o" and ".so" object files are regarded as specializations of class `Module`, thus enabling the use of comprehensive inheritance semantics [10] and type checking [11] in their composition.

6 A MOP for Scheme Compilers

Like the CLOS MOP, this MOP carefully chooses a useful portion of the internal functionality of a scheme compiler in order to provide the Scheme programmer with the desirable attributes of flexibility and control over layout and access over run-time data. Many of the details of this MOP are still under development [3, 12], so we only give a general description of it.

Unlike the CLOS MOP, this is a compile-time MOP, i.e. the accessible meta-architecture is specializable to control the *static* behavior of the compiler. Such static specialization is utilized at run-time. However, meta-code (the code that accesses the MOP) is not executed at run-time. This architecture decouples the language of the static processor (compiler), and hence the meta-architecture, from the source language itself — thus it is not meta-circular. The meta-architecture is expressed in an O-O extension of LISP called Traces [13]. This meta-architecture attempts to capture certain aspects, such as procedures and pairs, of a non O-O base language, Scheme.

The primary abstraction in this meta-architecture is what is termed a "contract." A *contract metaobject* represents a group of interrelated source program fragments that must agree on the layout of run-time data. For example, a `lambda` abstraction and all applications of (i.e. calls to) it would be such a group. Contracts essentially capture the notion that an abstraction and its uses must *statically* agree on conventions such as run-time layout. Other such static "contracts" include `cons` pairs along with its accessors `car` and `cdr`, and `let` environments along with their variable accesses.

Dependencies between abstractions and their uses are traced by flow analysis on source program fragments captured as *program graph metaobjects*. Source programs are translated into a register transfer language captured as *RTL metaobjects*. For instance, when a function application is required to generate code, it delegates the job to its contract metaobject, which further requests the appropriate program graph metaobjects to generate RTL metaobjects.

A few applications of the Scheme MOP are illustrated in [12]. These include extending the base Scheme language to support procedures with extra data attached to them, immutable data structures, and procedures that are dispatched based on the number of input parameters.

7 Summary and Conclusions

In this section, we attempt a summary of the meta-architectures surveyed. Of course, it is impossible to provide a comprehensive summary of the depths of the meta-architectures; instead we give a broad comparison of some essential aspects.

Figure 5 shows the abstractions, both O-O (e.g. class) and non-O-O (e.g. function), captured as meta-classes by the architectures. Figure 6 gives a comparative summary of the architectures in the areas of inheritance, method dispatch, encapsulation, and static typing.

	Smalltalk	CLOS MOP	Etyma	Scheme
Meta	`Metaclass`	(none)	(none)	N/A
Class	`Class`	`standard-class`	`Module/Interface`	N/A
Instance	`Object`	`standard-object`	`Instance/InstanceType`	N/A
Function	`BlockClosure`	(none)	`Function/FunctionType`	*lambda contract*
Variable	`Variable`	(none)	`Location/LocationType`	*let contract*
Primitive Value	`Magnitude`, etc.	(none)	`PrimValue/PrimType`	*[primitive] contract*

Figure 5: Summary of selected abstractions

Inheritance	In Smalltalk, the class `Behavior` and its subclass `Class` together model single inheritance semantics. In CLOS MOP, generic functions specialized to class `class` model multiple inheritance semantics. In Etyma, class `Module` implements unbundled inheritance operators. The default semantics of inheritance is significantly broader in Etyma/Jigsaw compared with the defaults in either Smalltalk or CLOS MOP.
Method dispatch	In Smalltalk, method dispatch is done by the `perform:...` method of class `Object`. In CLOS MOP, a discriminating function associated with class `generic-function` performs method dispatch. In Etyma, it is done by the method `select` of class `Instance`. CLOS MOP, by virtue of its very general model of generic functions and multi-methods, provides the most sophisticated method dispatch semantics.
Encapsulation	Smalltalk supports strong encapsulation of instance variables, and Etyma/Jigsaw encapsulates module attributes subjected to `hide` operations. CLOS MOP's default encapsulation semantics is weak, although metaobjects could be specialized using the MOP to support better encapsulation.
Static typing	Static typing and separate processing of modules are highly desirable attributes for languages. The Jigsaw language supports static type rules which have been incorporated into Etyma's `Interface` abstraction. Etyma also incorporates a comprehensive model of type meta-classes. Such a model is practically absent in the other meta-architectures.

Figure 6: Summary of O-O semantics

Although Smalltalk cannot boast generality in the area of inheritance, it still provides the most uniform and comprehensive model of concepts as objects in its meta-architecture.

It is also the most comprehensively designed meta-architecture, considering the complexity of the interacting dual hierarchies of meta-classes. The CLOS MOP, on the other hand, provides a pragmatic and systematically documented MOP, making it the most useful to applications. Etyma/Jigsaw provides significant generality compared with the other architectures, but its utility is yet to be demonstrated. The Scheme MOP is as yet experimental and in the process of being developed, but is unique and very promising.

Smalltalk is the clear winner in the area of abstractions for non O-O concepts in the language. The abstractions are general, broadly conceived, and uniform. The Scheme MOP attempts to capture only those basic concepts in the language which are important from a compilation standpoint. Etyma is currently attempting to design general abstractions covering the space of basic values and types in some commonly found languages.

In conclusion, meta-architectures are powerful, flexible, and extensible by their very nature. There are considerable similarities and differences in the goals that the architectures surveyed here are trying to achieve, as well as in their conceptions. Each was designed with a different set of requirements, yet they can be used for similar applications. The space of meta-architectures span from the pragmatic to the very general. In this paper, we have surveyed the goals, semantic models, and applications of four meta-architectures — Smalltalk, CLOS MOP, Scheme compiler and Etyma — and highlighted their salient features.

Acknowledgments

We gratefully acknowledge useful discussions and input from Gregor Kiczales, Bob Kessler, and Tim Moore.

References

[1] Johnson, R. E. and Russo, V. F., (1991), "Reusing object-oriented designs," Tech. Rep. UIUCDCS 91-1696, University of Illinois at Urbana-Champagne.

[2] Kiczales, G., des Rivières, J., and Bobrow, D. G., (1991), *The Art of the Metaobject Protocol.* Cambridge, MA: The MIT Press.

[3] Lamping, J., Kiczales, G., Rodriquez, L., and Ruf, E., (1992), "An architecture for an open compiler," in *Proc. of the IMSA '92 Workshop on Reflection and Meta-level Architectures.*

[4] Goldberg, A. and Robson, D., (1983), *Smalltalk-80: The Language and its Implementation.* Addison-Wesley.

[5] Bracha, G. and Lindstrom, G., (1992), "Modularity meets inheritance," in *Proc. International Conference on Computer Languages*, (San Francisco, CA), IEEE Computer Society, pp. 282–290. Also available as Technical Report UUCS-91-017.

[6] Bracha, G., (1992), "The programming language *jigsaw*: Mixins, modularity and multiple inheritance,". PhD thesis, University of Utah. Technical report UUCS-92-007; 143 pp.

[7] Keene, S. E., (1989), *Object-Oriented Programming in Common Lisp.* Reading, MA: Addison-Wesley.

[8] Lee, A. H., (1992), "The persistent object system MetaStore: Persistence via metaprogramming,". PhD thesis, University of Utah. Technical report UUCS-92-027; 171 pp.

[9] Banavar, G. and Lindstrom, G., (1993), "A framework for module-based language processors," Computer Science Department Technical Report UUCS-93-006, University of Utah.

[10] Orr, D. B. and Mecklenburg, R. W., (1992), "OMOS — An object server for program execution," in *Proc. International Workshop on Object Oriented Operating Systems*, (Paris), IEEE Computer Society, pp. 200–209. Also available as technical report UUCS-92-033.

[11] Banavar, G., Lindstrom, G., and Orr, D., (1994), "Type-safe composition of object modules," in *Computer Systems and Education: In honour of Prof. V. Rajaraman*, pp. 188–200, Bangalore, India: Tata McGraw Hill Publishing Company, Limited. ISBN 0-07-462044-4. Also available as Technical Report UUCS-94-001.

[12] Kiczales, G., Lamping, J., and Mendhekar, A., (1994), "What a metaobject protocol based compiler can do for lisp." Unpublished report. A modified version to be presented at the OOPSLA '94 workshop on O-O Compilation.

[13] Kiczales, G., (1993), "Traces (a cut at the "make isn't generic" problem).," in *Proc. of Int'l Symposium on Object Technologies for Advanced Software*, vol. 742 of *Lecture Notes in Computer Science*, Springer Verlag.