

Balancing the EULISP Metaobject Protocol*

HARRY BRETTHAUER[†]

(*bretthauer@gmd.de*)

JÜRGEN KOPP

(*kopp@gmd.de*)

*German National Research Center for Computer Science (GMD),
P.O. Box 1316, W-5205 Sankt Augustin 1, FRG*

HARLEY DAVIS

(*davis@ilog.fr*)

ILOG SA., 2 avenue Galliéni, 94253 Gentilly, France

KEITH PLAYFORD

(*kjp@maths.bath.ac.uk*)

School of Mathematical Sciences, University of Bath, Bath BA2 7AY, UK

Keywords: Object-oriented Programming, Language Design

Abstract. The challenge for the metaobject protocol designer is to balance the conflicting demands of efficiency, simplicity, and extensibility. It is impossible to know all desired extensions in advance; some of them will require greater functionality, while others require greater efficiency. In addition, the protocol itself must be sufficiently simple that it can be fully documented and understood by those who need to use it.

This paper presents the framework of a metaobject protocol for EULISP which provides expressiveness by a multi-leveled protocol and achieves efficiency by static semantics for predefined metaobjects and modularizing their operations. The EULISP module system supports global optimizations of metaobject applications. The metaobject system itself is structured into modules, taking into account the consequences for the compiler. It provides introspective operations as well as extension interfaces for various functionalities, including new inheritance, allocation, and slot access semantics.

While the overall goals and functionality are close to those of Kiczales *et al.* [10], the approach shows different emphases. As a result, time and space efficiency as well as robustness have been improved.

*This article is a revised and extended version of [5]

[†]The work of this paper was supported by the joint project APPLY, Ilog SA, the University of Bath, the British Council/DAAD ARC program, and the EULISP working group.

The joint project APPLY is funded by the German Federal Ministry for Research and Technology (BMFT). The partners in this project are the University of Kiel, the Fraunhofer Institute for Software Engineering and Systems Engineering (ISST), the German National Research Center for Computer Science (GMD), and VW-Gedas.

1. Introduction

Recently, object-oriented languages with metaobject protocols have begun to gain acceptance. A metaobject protocol extends the default semantics of an object-oriented language with an open, documented protocol, allowing the programmer to extend the base language in directions appropriate for his application. Instead of bending the application to fit the language, the programmer bends the language to fit the application. Ideally, many such extensions can peacefully coexist within the same basic framework; the language will treat the extensions homogeneously.

Additionally, metaobject protocols can provide generalized reflective facilities which allow the construction of debugging environments, inspectors, and other tools which manage all objects via the same set of operations.

The state of the art in metaobject protocol design is best described in “The Art of the Metaobject Protocol” by Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow [10]. Indeed, it is still an art rather than a science to define elegant and useful object-oriented programs, and the problem is compounded for a program as general as a language. Kiczales *et al.* present an elaborate and tested metaobject protocol (MOP) for the Common Lisp Object System (CLOS) [13]. Furthermore, they introduce the essential problems to the reader and show various techniques which can be used to solve them. Open questions and unsolved problems are presented to direct future work.

One of the main problems is to find a better balance between expressiveness and ease of use on the one hand, and efficiency on the other.

Since 1989, the authors of this paper and other members of the EULISP committee have been engaged in the design and implementation of an object system with a metaobject protocol for EULISP [12] intended to correct some of the perceived flaws in CLOS, to simplify it without losing any of its power, and to provide the means more easily to implement it efficiently. The current status of this work is reflected in the EULISP definition. The object system, TEAOΣ, has been implemented with minor variations in the public domain EULISP implementation FEEL¹, in the commercially available dialect LE-LISP VERSION 16 [1], in COMMON LISP¹ [2], and in SCHEME¹. TEAOΣ is used as the base for a set of artificial intelligence and graphic programming tools marketed by Ilog, SA. The expert system workbench *babylon* [7] marketed by VW-Gedas uses MCS² [3] which is closely related to TEAOΣ. Most of these tools extend the kernel object language provided by TEAOΣ using the metaobject protocol.

¹All three implementations are available via anonymous ftp from host ftp.bath.ac.uk in the directory /pub/eulisp.

²MCS is available by anonymous ftp from /lang/lisp/mcs on ftp.gmd.de

This work builds not only on CLOS, but also on a series of European work on simple reflective object-oriented architectures in LISP, including work on OBJVLISP [8], the Micro Flavor System [6], Micro Common Flavors [9], and the Meta Class System (MCS) [3].

1.1. Design Context

TEAOΣ is an integrated part of the EU LISP language definition. In describing it, we cannot completely isolate the object system from the rest of the language. There is a strong synergy between the rest of the language and the object system, especially in the diverse ways that software engineering goals are supported. For example, the division of work between classes and modules is discussed in more detail below.

EU LISP's primary goal is to serve as a general programming language offering the traditional power of LISP while taking the best concepts from other languages and striving for the possibility of simple, efficient implementations. EU LISP features the following essential elements:

- a module system to support separate compilation and encapsulation.
- division into a core language and libraries to facilitate small application development.
- parallel processes based on threads and semaphores for modern and future computer architectures.
- a condition system for error handling.
- downward continuations for flexible control structures.
- macros for syntactic extension.
- an object system based on classes and generic functions with simple default behavior and a metaobject protocol.

All elements of the language, except modules, bindings and the predefined syntax (defining and special forms), are represented by first-class objects. It is impossible to create or change a binding by computing its identifier at runtime. Although functions can be generated dynamically at runtime, all code patterns are known at compile-time. All of this helps generate small, efficient applications, allowing EU LISP to compete favorably with more traditional languages.

1.2. Design goals

Certain design goals apply almost to all languages and systems. These include robustness, abstraction, extensibility, ease of use, and efficiency.

Kiczales *et al.* [10] claim to meet a number of these important design criteria. Why is it hard or even impossible to meet all of them? The criteria are, in practice, contradictory if they must all be met simultaneously. However, we can use the fact that the priority of the goals change during the course of the software lifecycle to emphasize the most important goals at each lifecycle phase, thereby reducing design goal conflicts. For example, abstraction and ease of use apply mainly to the development and maintenance phases, while efficiency is essentially a runtime goal.

We propose the following classification of the above goals:

1.2.1. *Robustness*

The programmer must be able to depend on the documented functionality of defined modules; in other words, their semantic integrity must be enforced. Adapting them should not mean changing them, since other modules used in the same system might stop working as a consequence. CLOS, and COMMON LISP generally, violate this constraint through various redefinition facilities. Instead, we should distinguish between development time and execution time. During development, dynamic redefinition is useful. At execution time, however, the semantics of language entities should be fixed. Since the development environment is not generally considered to be appropriate for specification, we have little to say about the extended development capabilities offered by implementations. However, we would like to assure the programmer that he can write programs whose semantics is well-defined.

1.2.2. *Abstraction*

There should be different levels of granularity in the protocol, reflecting the different levels at which users need to think about the system's functionality and extend it. Not all extensions are equal. Some extensions require only small modifications in behavior from the default, while others are quite large. The amount of work required to implement an extension should reflect its scale. Although the user should not have to know implementation details, our experience shows that revealing appropriate details at appropriate levels often makes use easier. A delicate question is the lowest level of detail which can be revealed to allow portable and efficient implementations.

1.2.3. *Extensibility*

The scope of object models and implementations currently supported by various languages and tools is quite large, ranging from classless, prototype-based systems like Self to the complex, intricate models supported by many expert system tools. If our goal is to provide a single medium in which all of these models can harmoniously co-exist, special care must be taken in the development of flexible, general protocols.

This goal also interacts with robustness. We consider extension rather than modification to be the appropriate model for implementing new system behavior, since it allows both the definition of new functionality and constant semantics for existing functionality.

1.2.4. *Ease of use*

Using and extending the language must be natural and straightforward. Ease of use should not, however, be confused with the laziness of programmers who write quick hacks with unpredictable consequences. The fact that `defgeneric` is optional in CLOS is an example for such a doubtful support, in our view.

1.2.5. *Efficiency*

MOP-based systems are intended for large and serious software projects; the extensive freedom of a MOP cannot (and should not) be perceived by small applications. Using a metaobject protocol appropriately should increase the overall efficiency of complex systems. Efficiency has the highest priority at execution time and programs should follow the principle “don’t use, don’t lose”. Time and space are both important: As the power of our hardware increases, so too does the ambition of software developers. We do not agree with the dictum that “efficiency will not be a problem with next year’s computers”.

However, since efficiency concerns are especially vulnerable to conflict with other goals, they must be emphasized in the appropriate place. Here we make the sweeping generalization that compile-time and load-time efficiency are less important than run-time efficiency. In order to achieve high efficiency and extensibility, we try to put the high-cost operations at load-time. Future research will be directed in putting more effort into compile-time extensions.

All the above goals can be structured to reflect the two sides of a computer system. A computer language is an intermediary between a human and a machine; both have different needs which the language must try to balance. It must provide both:

- Support for good software engineering practice; this reflects the aspect *human* \Rightarrow *language*.
- Support for efficient programs; this reflects the other aspect, *language* \Rightarrow *machine*.

In the following, we show how the TEAOΣ MOP tries to achieve both of these goals. Note that the balance we describe is somewhat different than that noted by Kiczales *et al.* [10], where the tension is between flexibility and efficiency. Here, we broaden flexibility to include general software engineering goals, since the metaobject protocol is a part of a programming language.

1.3. Design Approach

By applying the following rules we hope to achieve the above goals:

- Simpler is better (instead of “worse is better” vs. “right is better”).
- Orthogonal language constructs are better.
- Development and execution requirements should be distinguished.
- A module’s compile-time and run-time dependencies should be distinguished.
- Efficiency costs should be paid at load-time rather than run-time.
- Language support is needed for a clear separation between extension definition and extension use.
- Restrictions due to efficiency concerns should be made explicit in the language, rather than in the documentation.

These rules influence software engineering as well as efficiency. The first two rules warrant further explanation; they influence each other and their violation often arises from a single cause. To determine which constructs should be provided by a language, we must identify the problem solving methods and paradigms used by humans. Different methods should be supported by orthogonal constructs which can then be kept simpler.

A good computer language should have a simple efficiency model. That is, constructs in the language should map simply onto implementations. The TEAOΣ slot access protocol is an example of a simple efficiency model, whereas the CLOS protocol (and COMMON LISP in general) explicitly rely

on clever implementations for achieving efficiency. Clearly, efforts in this direction must be continual, and we do not claim that TEAOΣ is the last word in this evolution. Rather, it points a direction in which further work can be done.

TEAOΣ could be viewed as a simplified version of CLOS and the CLOS MOP. However, we claim that the result is more powerful and more appropriate for most users based on our experience with MCS, FEEL and LE-LISP VERSION 16. The proof, of course, must still be provided by the experience of a wider range of users. The various public domain and commercial versions of TEAOΣ are starting to provide that feedback now.

2. Modular Decomposition

Many language designers believe classes and modules serve similar or identical purposes [11]. However, this belief is not universally shared [14]. The class/object construct provides data abstraction with specialization and generalization of structure and behavior of object classes; in contrast, the module construct deals with scope and extent of variable bindings and import/export relationships between modules supporting information hiding and encapsulation. Classes serve primarily to model the problem domain, while modules aid problem decomposition. Another way to look at the distinction is to think of classes as *implementation* devices and modules as *interface* devices.

COMMON LISP does not provide support for strict import, export, and visibility aspects. Its package system considers symbols as subjects of exchange. Symbols, however, are used for many binding spaces: global functions, variables, classes, types, and so on. Thus, exporting a symbol for one purpose opens the door automatically for all the others.

Languages like C++ or Eiffel overload classes by import/export and visibility features. That makes their class concept as well as their scope rules complex, especially when inheritance comes into play.

In EuLISP, we use the distinction between classes and modules to provide a module system orthogonal to the object system, thus supporting better software engineering practice as well as better efficiency.

The example in Figure 1 hints at how the distinction between compile-time and runtime dependencies can be expressed by the programmer. Up to now, the EuLISP committee has only specified the semantics of importing macros as compile-time imports. However, compile-time imports can also be applied to other meta-level features like metaobject classes and their operations. Using this information, the compiler could more easily decide which optimizations to apply.

```

(defmodule non-reflective-object-system
  ;; interface
  (import (primitive-language-elements
           reflective-object-kernel)
         syntax (comfortable-syntax)
         export (defstruct
                 defgeneric defmethod
                 generic-lambda
                 call-next-method next-method-p
                 make initialize allocate))
  ;; implementation
  ...)

(defmodule non-reflective-application
  ;; interface
  (import (non-reflective-object-system ...)
         syntax (comfortable-syntax
                 non-reflective-object-system-syntax)
         export (start-application))
  ;; implementation
  (defun start-application () ...) ...)

```

Figure 1: Two example module definitions

A module can be compiled separately, generating a library, or an entire application can be completely compiled, including all of its imported modules, generating a stand-alone application. While many global optimizations are difficult and unsafe in COMMON LISP EULISP provides direct support for making them straightforward and based on clear semantics.

2.1. Structuring TELOS Using Modules

The different parts of the MOP are separated into modules to gain clarity and efficiency for applications.

We want to have a simple module of object-oriented constructs which can be analyzed statically allowing significant optimizations by the compiler when used in applications. In particular, all classes, generic functions and methods should be known at compile-time. We must keep the mass of a complex application non-reflective in order to achieve the same performance as in non-reflective languages. Furthermore, programs are more understandable if reflective and non-reflective parts are clearly separated.

In the above example we show, through the export list of the module `non-reflective-module`, which language constructs we consider as non-reflective. These are the defining and anonymous creation forms for classes, generic functions, and methods, as well as the instance creation and initialization functions `make`, `initialize`, and `allocate`.

The other modules can be divided in those allowing introspection and those allowing specialization of special kinds of metaobjects like classes, slot descriptions, generic functions, and methods. The introspection modules export the corresponding classes and slot readers. The specialization modules additionally export the operations specified by the initialization protocols. Furthermore, we provide a module exporting portable low level allocation primitives.

3. The Metaobject Protocol

The following sections summarize the salient points of the TEAOΣ metaobject protocol as it reflects the design philosophy described above. We assume the reader to be familiar with the CLOS MOP to contrast the relevant aspects of the two protocols.

The slot access model is described in detail to illustrate the general principle of moving as much work as possible to load-time. The higher order capabilities of Lisp are exploited by protocols which compute functions to be used at runtime – a kind of configurable dynamic compilation process. By closing over all precomputable information, we can avoid a great deal of runtime work.

The protocols controlling instantiation and inheritance are described only briefly, highlighting mainly their simpler default behavior when compared with their analogs in CLOS. Note that some differences between the TEAOΣ and CLOS protocols go unreported here since they are beyond the immediate scope of this paper.

3.1. Slot Access

The slot access model adopted within TEAOΣ departs from CLOS. Before explaining the new protocol we will first justify it by identifying the features of the CLOS approach which put it at odds with our stated design philosophy.

3.1.1. The CLOS Approach

From our design standpoint, the CLOS slot access protocol is not a reasonable solution. Although it provides for straightforward extension of the default slot access behavior, the following properties present problems:

Inherent inefficiency Slot access time is crucial to the performance of object-oriented applications. Recognizing this, a simple efficiency model (as defined above) is desirable.

The primary route to the value of a slot of a CLOS object is through the `slot-value` chain. This is a *dynamic protocol*, a MOP protocol that must be honored at runtime – even, in principle, in a non-reflective application – with all of the attendant runtime overhead³. To achieve acceptable performance, CLOS relies on implementations to circumvent this route whenever possible while still honoring new methods added to the protocol functions. Typically, accessor functions are optimized in some way, often via some new protocol for their computation, leading to further problems.

Competing protocols Problems of consistency can often arise between computed accessors and the dynamic protocols due to the instability of the complex optimizations being employed. The existence of these two methods of slot access also raises uncertainty as to which is used by other areas of the MOP such as initialization.

So, although the CLOS slot access protocol provides a flexible means of extension, the cost in terms of the complexity of efficient, consistent implementations is too great. The design goals for TELOS suggest that it should be replaced by a protocol which, while retaining flexibility and simplicity, maps more naturally to reasonable implementations.

3.1.2. *The TELOS Approach*

Rather than have a dynamic slot access protocol, TELOS provides a standard protocol for computing readers and writers. Every slot description contains one reader and one writer capable of extracting and updating the corresponding slot within instances. Slot options in `defclass` which define accessors merely bind the slot's single reader or writer to the appropriate name. Therefore, two readers for the same slot bound to different names will always be `eq`.

Orthogonality of design is maintained by describing *all* slot accesses as taking place through calls to these accessor functions. No analog of `slot-value` is provided⁴.

³Every slot access requires at least one standard function call, two generic function calls and a list or table lookup for the appropriate slot definition object (ignoring the cost of accessing the set of slot definitions from the class object).

⁴Although it can be written simply in terms of accessors.

3.1.3. The Slot Access Protocol

Accessors are computed and updated as part of the initialization of a class. A new slot, inherited from no superclass, has a fresh reader and writer computed for it. These functions are then stored in its slot description. The protocol generic functions

```
COMPUTE-SLOT-READER class slotd slotds
COMPUTE-SLOT-WRITER class slotd slotds
```

are used to compute these functions. Accessor functions computed in this way are not guaranteed to work for direct instances of a particular class unless they have been *ensured* for that class. Typically, `compute-slot-reader` will return a generic function without any methods defined on it.

Inherited slots, either specialized in some way or left unchanged, take the reader and writer from the corresponding slot description objects of the superclasses. To combine two or more inherited slot descriptions, they must have a common root and thus share the same accessor functions. Note, then, that there is a one-to-one correspondence between logical slots and their accessor pairs.

Before inheritance is complete, the accessors of the slot descriptions of the class must all be ensured. The protocol generic functions

```
ENSURE-SLOT-READER class slotd slotds reader
ENSURE-SLOT-WRITER class slotd slotds writer
```

are called to guarantee that the accessors will work on direct instances of the new class. Typically, `ensure-slot-reader` will add a method to `reader` capable of reading the appropriate slot of direct instances of `class`. In cases where the slot has not “moved” relative to its position within instances of the superclasses of `class`, there may be no need to update the reader.

The computed accessor protocol also gives a portable way of defining lower overhead slot accessors by making readers and writers standard functions where appropriate:

```
(defmethod compute-slot-reader
  ((c <structure-class>) (sd <slot-description>) slotds)
  (compute-primitive-reader-using-class c sd))

(defmethod ensure-slot-reader
  ((c <structure-class>) (sd <slot-description>) slotds reader)
  reader)
```

Slots of structure classes, which always use single inheritance, never change position in subclasses and the accessors should not check the types of their arguments. The above methods implement this functionality for readers. A primitive reader is returned as the slot accessor – `ensure-slot-reader` need not do anything since the slot position can never change.

The standard ensuring methods use a subprotocol for computing *primitive accessors* – standard functions capable of accessing a particular slot in direct instances of a given class. These protocol functions are the direct analog of the `slot-value-using-class` tier in CLOS and are the generic functions most commonly used to change the behavior of slot access.

```
COMPUTE-PRIMITIVE-READER-USING-SLOT-DESCRIPTION slotd class slotds
COMPUTE-PRIMITIVE-WRITER-USING-SLOT-DESCRIPTION slotd class slotds
COMPUTE-PRIMITIVE-READER-USING-CLASS class slotd slotds
COMPUTE-PRIMITIVE-WRITER-USING-CLASS class slotd slotds
```

For example, `compute-primitive-reader-using-class` returns a function of one argument that when applied to a direct instance of `class`, returns the value of the slot described by `slotd`. Its behavior on instances of other classes, even subclasses of the specified class, is undefined.

3.1.4. Comparing Use

From the point of view of the applications programmer, there is little practical difference between the CLOS and TELOS slot access protocols. The functional equivalence between the two can be illustrated by a simple example. Let's say we want to implement a new slot description which verifies that a slot value, when set, matches a certain predicate. In both CLOS and TELOS, we can store the predicate in the slot description object. The difference lies in how to specify the behavior of writing to such a slot.

Here is the CLOS method:

```
(defmethod (setf slot-value-using-class)
  (new-value (class standard-class)
    object (slot predicate-slot-definition))
  (assert (funcall (slot-definition-predicate slot) object))
  (call-next-method))
```

Here is the equivalent TEAO Σ method:

```
(defmethod compute-primitive-writer-using-slot-description
  ((slot <predicate-slot-description>) (class <class>) slotds)
  (let ((prev-writer (call-next-method))
        (predicate (slot-description-predicate slot)))
    (lambda (object new-value)
      (assert (predicate object))
      (prev-writer object new-value))))
```

We can note that the TEAO Σ method is slightly more complicated, but more efficient since it accesses the predicate just once compared to on every write as in the CLOS case. In this small example, the effect of this is minimal, but it can be quite significant in others. Also note that the function returned by the TEAO Σ method is called directly by the writer generic function.

All primary methods for `slot-value-using-class` could be translated mechanically into `compute-primitive-reader-using-class` methods.

In cases where it might be desirable to use a dynamic protocol, it is a simple matter to specialize `compute-slot-reader` to return a standard closure which honors that new protocol. Similarly, it should be possible to implement the computed protocol as an extension to CLOS but this would be somewhat more involved.

3.1.5. Implementation and Efficiency

Despite an increase in the number of functions over its dynamic counterpart, implementation of the computed TEAO Σ protocol is no more difficult – a naive implementation need take no more than a few lines of obvious code to implement each function in the protocol.

The clearest difference in terms of runtime efficiency is that the computed accessor protocol reduces slot access in standard classes to a single generic function call in even the simplest implementation. This has been done without loss of generality or significant reduction in the ease with which the behavior of the object system may be extended. The uniform use of accessors also ensures that this potential improvement is propagated into other areas such as initialization.

In addition, one expected effect of reducing the relative significance of slot access as a performance bottle-neck is to allow implementors to concentrate their optimization efforts on a smaller set of “hot spots”.

3.2. Method Lookup and Generic Dispatch

TEAO Σ uses the generic function mechanism introduced by CLOS to

implement polymorphic behavior. However, the default generic function mechanism of TEAOΣ is simplified compared to CLOS; rather than introducing dubious and costly extensions in the kernel, we choose to relegate certain functionality to extension modules, and provide enough extensibility to allow portable versions of them to be written.

For example, the default generic function dispatch in TEAOΣ is purely class based; `eq1` methods are not supported. However, `eq1` methods like those found in CLOS can be portably implemented in an extension module by defining a new generic function class and new methods on the method lookup generic function.

3.3. Allocation and Initialization

In the current specification of TEAOΣ, the instance creation protocol is very similar to that of CLOS. However, we intend to apply the load-time priority principle to instance creation as well. Classes will then have associated allocator and initializer methods computed for them at class creation time. Extensions to allocation or initialization will thus be done by extending the allocator or initializer-generating methods rather than some general allocating or initializing generic function, as is done in CLOS and the current version of TEAOΣ.

3.4. Class Definition and Inheritance

TEAOΣ supports a standard inheritance protocol as flexible as that proposed by Kiczales *et al.* [10] and of a slightly finer granularity, splitting the work into a number of explicit phases.

The move towards a more load-time weighted protocol translates into more work being done at class instantiation time. This work includes computing and ensuring accessors, allocators and initializers as described above. The generation of these functions constitutes a phase in itself.

The default inheritance methods implement single inheritance. The protocol, however, is designed so that general multiple inheritance or mixin inheritance [4] can be easily and portably implemented in extension modules⁵.

Neither class redefinition nor changing the class of an instance is supported by standard classes. It is this, in combination with the guarantee that the behavior of standard generic functions cannot be modified for standard classes⁶, which imbues programs expressed in terms of the default

⁵It is likely that all three kinds of inheritance will be supported by standard EU LISP library modules.

⁶Due to the absence of support for method removal and non-standard method combination.

metaobjects with static semantics.

Class, method, and generic function redefinition as well as class change may all be portably implemented as library extensions as has been done in MCS. They are also desirable features of an interactive development environment. We envisage that such development environments may transparently, by simple module substitution, allow code to be developed in terms of a set of metaobjects supporting these facilities in place of the standard metaobjects. If necessary, to gain speed or space efficiency, the program may be recompiled in terms of the standard metaobject set without change.

4. An Example – Mixin Inheritance

This section sketches an extension of the TEAOΣ MOP which implements mixin inheritance. Since only the slot access protocol has been described in detail in this paper, we focus on that aspect here.

4.1. Informal Specification

The goal of mixin inheritance is to provide a more expressive and flexible programming style than single inheritance while avoiding certain problems associated with general multiple inheritance. Mixin inheritance distinguishes between essential and subsidiary properties of objects when classifying them in a problem domain. We associate *base classes* with essential properties and *mixin classes* with subsidiary properties. From the modeling point of view, essential properties are substantive — such as buffers and windows. Subsidiary properties are descriptive — such as printable, bordered, and titled.

Mixin inheritance clarifies class hierarchies and improves application efficiency by obeying the following restrictions:

- A base class can directly specialize many mixin classes but only one base class. A super-base-class is considered more general than the super-mixin-classes.
- A mixin class can specialize many mixin classes but no base classes.
- There are no join nodes in the inheritance hierarchy, except the root class `<object>`.
- Base classes may have direct instances, while mixin classes may not be instantiated.

4.2. Implementation Outline

First, we have to define the two new metaclasses `<base-class>` and `<mixin-class>`. These metaclasses are the classes of the new kinds of classes we described above.

```
(defclass <base-class> (...) ())
```

```
(defclass <mixin-class> (...) ())
```

The exact superclasses are not specified here, but they must be metaclasses, and thus subclasses of `<class>`.

In this paper we concentrate on the slot reader generating protocol. However, the other generic functions in the inheritance protocol, which need to be defined for the new metaclasses, should be mentioned, too. These generic functions include:

```
compatible-superclasses-p,  
compatible-superclass-p,  
compute-class-precedence-list,  
compute-inherited-slot-descriptions,  
compute-slot-descriptions, and  
compute-specialized-slot-description.
```

The methods checking the compatibility of superclass and superclasses control the first and second restriction in the list given earlier. The method on `compute-class-precedence-list` linearizes the class hierarchy for base and mixin classes *depth first left to right*. It signals an error if a join node different from the root class `<object>` occurs. The method computing inherited slot descriptions returns a list containing the effective slot description lists of all direct superclasses. The new methods for computing slot and specialized slot descriptions have to deal with the case of multiple inherited slot descriptions with the same name. Often, we can reuse the system defined method via `call-next-method` and extend it as needed.

Now, we consider the slot accessor computation. Slots of base classes never change position in subclasses. Slots of mixin classes, however, can change position in subclasses. Thus, a generic function can be returned as the reader for slots defined by mixin classes, while a simple function is returned as the reader for slots defined by base classes.

```
(defmethod compute-slot-reader  
  ((c <base-class>) (sd <slot-description>))  
  (compute-primitive-reader-using-class c sd))
```

```
(defmethod compute-slot-reader
  ((c <mixin-class>) (sd <slot-description>))
  (generic-lambda ((o c))))
```

The `ensure-slot-reader` method does not need to do anything for mixin classes since they have no direct instances, and may rely on the assumption that the right thing will be done when a slot defined for a mixin class is inherited by a base class: A method applicable for the direct instances of the base class will be added to the generic reader.

```
(defmethod ensure-slot-reader
  ((c <mixin-class>) (sd <slot-description>) reader)
  reader)

(defmethod ensure-slot-reader
  ((c <base-class>) (sd <slot-description>) reader)
  (if (generic-function-p reader) ; mixin slot reader?
      (let ((r (compute-primitive-reader-using-class c sd ...)))
        (add-method reader (method-lambda ((o c)) (r o))))
      reader))
```

The slot writers are treated in an analogous way.

5. Conclusion

We have discussed the design goals and approach of the TEAOΣ metaobject protocol. They provide a better balance between support for good software engineering practice and support for efficient programs than the CLOS MOP.

We gave an overall description of TEAOΣ including a metaobject protocol which provides the openness needed for extensions and achieves efficiency by more static semantics for predefined metaobjects, modularizing their operations. Simplicity and orthogonality support both good software engineering practice as well as efficient programs. The metaobject system is structured into modules taking into account the consequences for the compiler. It provides introspection operations as well as extension interfaces for new inheritance strategies, new instance allocation methods, new slot descriptions, new slot access primitives, new discrimination methods, class redefinition, and so on.

With some minor variations, the major parts of the described protocol have been implemented in SCHEME, COMMON LISP, LE-LISP VERSION 16

and FEEL. While the general goals and functionality are almost the same as described by Kiczales *et al.* [10], the approach shows different emphases. As a result, efficiency as well as robustness have both been improved.

6. Acknowledgments

First, we want to thank Greg Nuyens and Julian Padget, the main editors of EULISP, as well as all members of the EULISP working group for the very fruitful discussions during meetings and on the net.

We thank Wolfgang Goerigk and Ingo Mohr for their suggestions from the APPLY compiler builders' point of view.

We thank the CLOS designers for providing an excellent and useful framework in which to study these issues, and for altruistically not doing everything right the first time.

We thank Gregor Kiczales, Jim des Rivieres and Daniel G. Bobrow for their excellent book "The Art of the Metaobject Protocol", and for making parts of the sources available to a wide community by ftp.

References

1. *Le-Lisp version 16 Reference Manual*. ILOG, SA, Gentilly (1992).
2. Bradford, R. Telos in Common Lisp. Submitted for publication (1993).
3. Bretthauer, H. and Kopp, J. *The Meta-Class-System MCS. A Portable Object System for Common Lisp. Documentation*. Arbeitspapiere der GMD 554, GMD, Sankt Augustin (July 1991).
4. Bretthauer, H., Christaller, Th., and Kopp, J. Multiple vs. Single Inheritance in Object-oriented Programming Languages. *Microprocessing and Microprogramming*, 28 (1989) 197–200.
5. Bretthauer, Harry, Davis, Harley, Kopp, Jürgen, and Playford, Keith. Balancing the eulisp metaobject protocol. In Yonezawa, Akinori and Smith, Brian C., editors, *IMSA'92 Workshop on Reflection and Meta-Level Architecture*, Tokio (1992) 113–118.
6. Christaller, Th. Eine Einführung in LISP. In Christaller, Th., Hein, H.-W., and Richter, M. M., editors, *Künstliche Intelligenz. Theoretische Grundlagen und Anwendungsfelder*, Springer-Verlag, Berlin (1988) 1–35.
7. Christaller, Th., di Primio, F., Schnepf, U., and Voß, A., editors. *The AI-Workbench BABYLON*. Academic Press, London (1992).

8. Cointe, P. The ObjVlisp Kernel: a Reflective Lisp Architecture to define a Uniform Object-Oriented System. In Maes, Pattie and Nardi, Daniele, editors, *Meta-Level Architectures and Reflection*, North Holland (1988) 155–176.
9. di Primio, F. *Micro Common Flavors*. Arbeitspapiere der GMD 295, GMD, Sankt Augustin (February 1988).
10. Kiczales, G., des Rivieres, J., and Bobrow, D. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts (1991).
11. Meyer, B. *Eiffel: The Language*. Prentice Hall Object-Oriented Series, Prentice Hall, New York (1992).
12. Padget, J. and Nuyens, G., editors. *The EuLisp Definition, Version 1.0*. In preparation.
13. Steele Jr., Guy L. *Common Lisp - The Language, Second Edition*. Digital Press, Bedford, Massachusetts (1990).
14. Szyperski, C. A. Import is Not Inheritance – Why We Need Both: Modules and Classes. In Madsen, O. Lehrmann, editor, *Proc. of the ECOOP '92*, Springer-Verlag (July 1992) 19–32.