Metalevel Solution to Inheritance Anomaly in Concurrent Object-Oriented Languages[*] — Extended Abstract —

Satoshi Matsuoka[†]
Akinori Yonezawa
Department of Information Science
The University of Tokyo

August 1, 1990

1 Introduction

1.1 OOCP, Reflection, and Professional Computing

In the very near future, massive parallel architectures will be available to professionals of numerous fields in the manner personal computers and workstations are today. *Professional computing* is the term we use for describing the computational activities of professionals requiring immense computational power. We claim that object-oriented concurrent programming (OOCP) serves as the basis for professional computing.

The ABCL project group, now at the University of Tokyo, is actively engaged in the research of OOCP[21]. A research conducted by Takuo Watanabe and Akinori Yonezawa of the ABCL project group has shown that meta-level architectures and computational reflection play significant roles for flexible modeling of activities of concurrent objects. In ABCL/R, a reflectional version of ABCL/1, examples were given for dynamic acquisition of methods, monitoring of objects, and the virtual time algorithm[18].

Fully-reflectional languages such as ABCL/R, however, tends to experience inefficiency due to their interpretive mode of execution. This is a severe disadvantage for concurrent computing, for substantial gain in the speed of computation is the primary motivation for resorting to concurrency in the first place. Fortunately, for many purposes, full computational reflection is not necessary; instead, 'partial' reflection will suffice. For example, [7] reviews the control-related meta-level facilities in LISP, such as `call/cc`. [9] demonstrates the effectiveness of partial reflection in Object-Oriented languages by extending the VM of Smalltalk so that message dispatching can be altered by the user. The Metaobject Protocol of CLOS is another example where partial reflection is effective[2].

[*](Submitted to the OOPSLA/ECOOP'90 Workshop on Reflections and Metalevel Architectures in Object-Oriented Languages)

[†]Physical mail address: 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan. Phone 03-812-2111 (overseas +81-3-812-2111) ex. 4108. E-mail: matsu@is.s.u-tokyo.ac.jp,yonezawa@is.s.u-tokyo.ac.jp

1.2 Dynamic Progression of the Degree of Reflectivity

Reflection is especially important in *OO-Concurrent* systems, where reflectional capabilities could be used for dynamic evolution of the system[20]. One of the current goals of the ABCL project is to build a portable software platform architecture with reflective capabilities in order to support dynamically evolving OOCP languages. The platform would serve as a basis for not only for ABCL but also for various other research in OOCP.

As our research proceeded, however, it became apparent that ad-hoc partial reflective capabilities were not powerful enough to simultaneously achieve the (somewhat contradictory) goals of speed/efficiency versus system flexibility/evolution — Since our reflection is partial, there is some degree of freedom as to how much of the system we could reify within itself. Then, there is a tendency that, the more we allow the computational structure to be reified/reflected, the efficiency of the system suffers; for example, it would be difficult to perform 'behind the back' optimization of objects. This especially poses difficulty in compiled code, where various code optimizations are natural occurences.

Our solution to this problem is to allow variance in the degree of reflection; we call our approach the *dynamic progression of the degree of reflectivity.* Reflection is 'lighter' when the system allows less of the structure/computation to be reified/reflected. The converse is 'heavier' reflection. By dynamically adapting to varying *degree of reflectivity* according to the necessity of the meta-level operation to be performed, we can accomplish both good performance and flexible system evolution.

2 The Conflict Between Synchronization and Inheritance

Before proceeding with describing the essential aspects of our proposal, let us first give an overview of the fundamental problem in OOCP we are attempting to solve with reflection.

When OOCP languages are put to use in the development of large-scale programs, one of the prime issues is synchronization of activities of objects in the system: When a concurrent object is in a certain state, it can accept only a subset of its entire set of messages in order to maintain its internal integrity. We call such a restriction on acceptable messages the *synchronization constraint* of a concurrent object. In most OOCP languages, the programmer gives either implicit or explicit program specification to control the set of acceptable messages. We call such specification the *synchronization specification.* The synchronization specification must always be *consistent with* the synchronization constraint of an object; otherwise the object might accept a message which it really should not accept, causing an error.

Another important facility of OO languages is *inheritance.* However, it has been previously pointed out that *inheritance* and *synchronization constraints* often conflict with each other[1, 3, 15]. Some have gone so far as not adopting inheritance in their languages[1, 17, 21, 22], or employed a flexible communication mechanism independent of the inheritance hierarchy[12]. Several proposals [4, 5, 6, 10, 14, 16] have been made in the past for controlling the anomaly arising from their simultaneous incorporation into OOCP languages. However, we can show that, for proposals which employ method keys (names) for synchronization specifications, the anomaly in inheritance occurs where *re-definitions of all relevant parent methods are necessary.* Our recent research [13] deals extensively with this subject.

We can also show that this anomaly can be avoided somewhat by attaching a predicate to each method as a guard for synchronization specification. This scheme, however still has several drawbacks:

1. naive implementation of guards is not very efficient, and

2. some classes of synchronization constraints (*trace-only sensitive*) cannot be effectively inherited.

The first problem can be partially resolved statically with the use of program transformation which is invisible to the programmer — we can obtain code that is near-optimal in the sense that the time efficiency of the code would match that of the code written with the previous proposals[13]. The second one is more serious: there are methods that have high generality, such as `:lock` and `:unlock`, yet are *trace-only sensitive* methods. Trace-only sensitive methods are methods whose synchronization constraint cannot be given as a first-order logical formula whose only allowed terms are either constants or instance variables of the object; terms such as message keys are not allowed. What this means is that guards alone are not sufficient for expressing the synchronization specifications of concurrent objects.

The ability to partially reflect upon the message processing of an object provides an elegant solution to such a problem. Furthermore, we show that, with dynamic progression of degree of reflectivity, it is possible to apply the solution without sacrifices in efficiency. We are currently designing a prototype reflective language, X0/R, whose purpose is to provide an efficient metalevel solution to the inheritance anomaly. The remaining sections of the paper will outline the reflective features of the X0/R and how it can be used to resolve the anomaly between the synchronization constraints and inheritance.

3 Reflective Features of Language X0/R

3.1 Brief Overview of Basic Syntax and Semantics

The syntax and the semantics of the language X0 is similar to that of ABCL/R. Each object consists of state (`instance-var`), message queue, and a set of methods (scripts)[1]. A method has an associated message pattern and a guard. Basically, objects communicate with one another solely by sending messages. There are two types of messages, the *past type* (`<=`) and the *now type* (`<==`). The received message is first placed in the message queue. When an object becomes ready to accept a message, it scans the queue for the first message which (1) matches one of the message patterns, and (2) the corresponding guard evaluates to `true`[2]. The receiver of the now type message returns a value with `[!`*value*`]`.

3.2 Informal Model of Object Reflection in X0/R

The model of reflection we currently employ in X0/R is basically that of languages such as ABCL/R or 3-KRS[11]; there is one corresponding meta-object per each object. Other alternatives could be considered, such as: (1) reflect upon a *system* of objects, and/or (2) have many-to-many relationships between the objects and meta-objects. The reasons for adopting the current approach are:

- Our prime objective is to realize an efficient metalevel architecture. Per-object meta-object is then preferable, as structural reflection of each object would allow us to optimize the behavior on a per-object basis.

[1]Note that methods and guards in X0/R are actually associated with the class objects and not instances themselves.

[2]This is different from ABCL/R, which always removes the FIRST message from the message queue, and performs pattern matching against the message patterns. The message which did not match any of the patterns is discarded — this prevents the violation of the *transmission ordering law*.

- Customization of the behavior of each object is effectively achieved with customization of its meta-object; we do not currently require the added complexity of many–to–many object–to–meta-object correspondences.

One of the prime difference between X0/R and ABCL/R is that we do not associate explicit evaluator of scripts with each meta-object; this is primarily due to the requirement that we run compiled code. Another difference between X0/R and ABCL/R is the presence of classes. Classes are themselves objects as in Smalltalk or CLOS, and are *totally distinct* from meta-objects. In essence, classes primarily act as prototypes or templates for instance creation, and instance behavior is more strongly governed by its meta-object. This is similar to the *specific meta-object model* as was categorized by [8]. With this approach, individual objects can dynamically alter their 'degree of reflectivity' without the necessity of changing their classes.

3.3 Lightweight Meta-Objects

We let the generic, top-level meta-object be the 'lightest' meta-object in the system, meaning that it is the least capable with respect to reflection, but is the most efficient with respect to code execution. When an instance is created with the default `new:` message to a class object, its meta-object, an instance of class `Lite-Meta-Object`, is created simultaneously and associated with the object[3]. All other 'heavier' meta-objects are instances of meta-object classes that inherit from `Lite-Meta-Object`. Below is an outline of the definition of `Lite-Meta-Object`; most methods are primitive methods defined by the system for efficiency:

```
[define-class Lite-Meta-Object
   [super Object]
   [instance-var
      (Class)           ;; class object of instance
      (Inst-Vars)       ;; associative vector of instance variables
      (Queue)           ;; message queue
      (Meta-of)         ;; meta-object of instance

   [primitive [:message M]   ;; dispatch method according to message
       ;; search message queue until a message is found
       ;; such that it matches a message pattern and the guard is true
       ;; The guard is found in the class object paired with the method.
       ;; Invoke the method.]
   [primitive [:get-all-guards] ;; collect and return all guards
       ;; collect the guards up the superclass hierarchy]
    ...
]
```

An object has a hidden `meta` field which maintains a link with its initial meta-object. Upon object creation, this field is set to point to its meta-object, and do not allow subsequent access or modification by the user program. The hidden `meta` field of the meta-object is initially `NULL` — by definition, an object whose `meta` field is `NULL` is considered to be an instance of class `Lite-Meta-Object`. This is to allow lazy creation of meta-objects to avoid infinite regression up the reflection tower.

[3]Notice that the `Lite-Meta-Object` class object itself is an object-level existence (Figure 1).

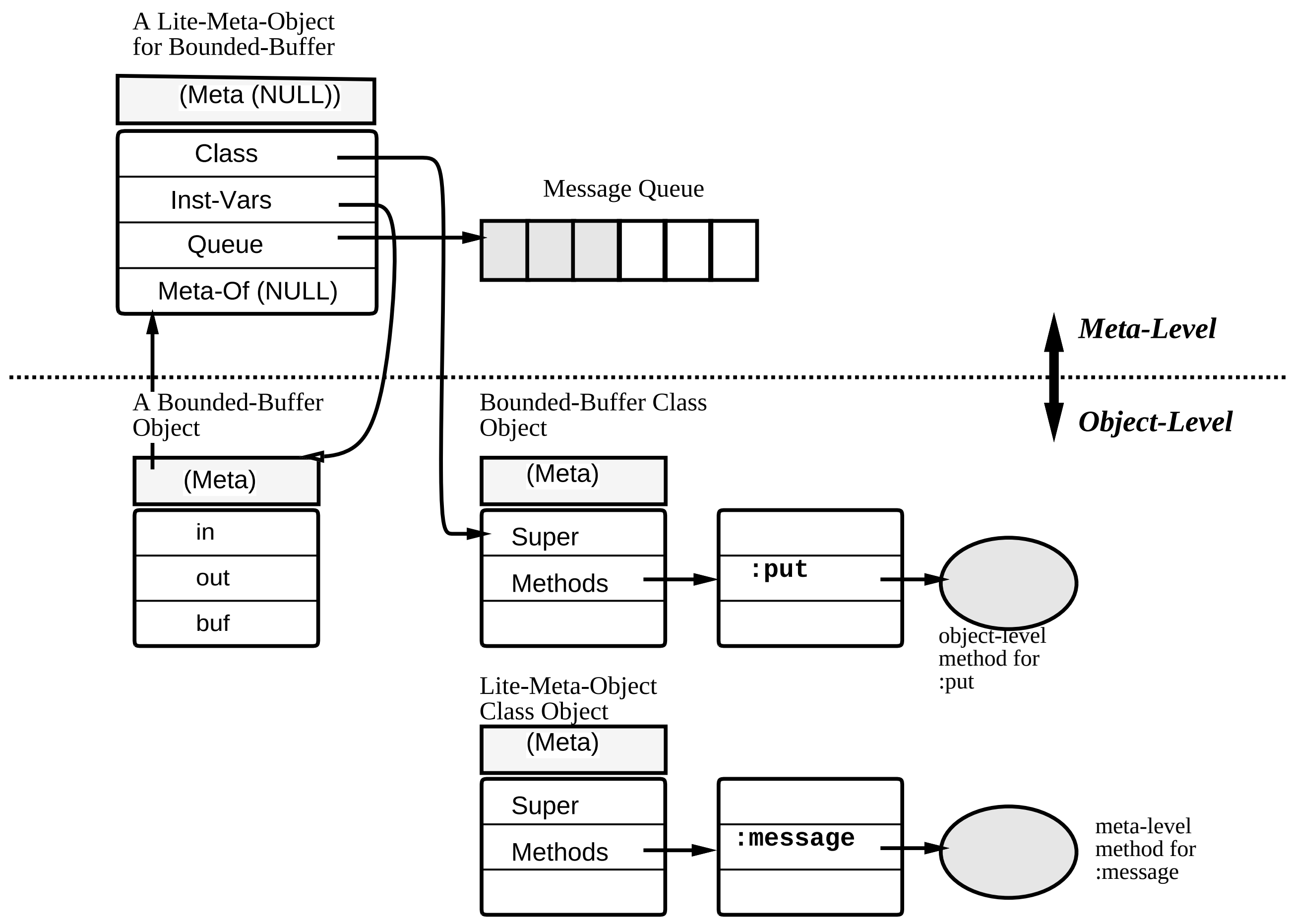


Figure 1: Initial Configuration Upon Instance Creation

There is an additional instance variable in the meta-object, `Meta-Of`, which holds a modifiable link to the meta-object of an object; this variable is also set to `NULL` on initial creation, indicating that the current meta-object matches that of the hidden link. The purpose for maintaining a separate variable in the meta-object is to easily maintain compatibility between the normal version of method lookup via the meta-object and the short-circuit version we discuss below. Figure 1 illustrates the configuration upon initial creation of instance of class `Bounded-Buffer`.

The method `:message` invokes the generic message handler of the system. Each message received by an object corresponds to the meta-level reception of the `:message` message by the meta-object. The system performs the following optimization to achieve the efficiency of direct message lookup: when the `Meta-of` of the meta-object is `NULL`, the system short-circuits the meta-level method lookup for `:message`, and directly invokes the default message handler. The method `:message` may be overridden by 'heavier' meta-object classes to customize the method dispatch; in such a case, the short-circuit optimization is naturally inactive.

3.4 Meta-Object Access and Lazy Meta-Meta-Object Creation

There are two special forms to go up/down the reflective tower as in ABCL/R: `[meta` x`]` and `[den` x`]`. For a given object x, let $x\uparrow$ be the meta-object of x. Then, `[meta` x`]` $\equiv x\uparrow$, and `[den` x`]` is its inverse, i.e. for a given y such that $y = x\uparrow$, `[den` y`]` $\equiv x$. `[meta` x`]` could be regarded as being equivalent to the following definition:

$$\texttt{[meta } x\texttt{]} \equiv \texttt{[}x\texttt{ <== [:meta]]}$$

where `:meta` is a generic method defined at the top class in the class hierarchy, `Object`:

```
[public [:meta]
   [temporary My-Meta]
   (set! My-Meta [[meta Self] <== [:get-meta-of]])
   (if (null? My-Meta)   ;; if   Meta-Of field is NULL
      (begin             ;; then set it to the current meta-object
         [[meta Self] <== [set-meta-of! [meta Self]]]
         [! [Self <== [:meta]]]
      [! My-Meta]])]     ;; else return the current value of Meta-Of
```

The reader may have already noticed that the definition of `[meta` x`]` is cyclic — in practice, the system 'cheats' and returns the value of the hidden `meta` field when appropriate.

The meta-object of a meta-object (the meta-meta-object) is created lazily when `[meta ...]` is evaluated. After its creation, the meta-object attains the ability to receive meta-level messages. This guarantees that the meta-object will be able to receive meta-level messages when necessary, for it is impossible to designate a meta-object as a target of the message otherwise (Figure 2).

4 Metalevel Solution to Inheritance Anomaly

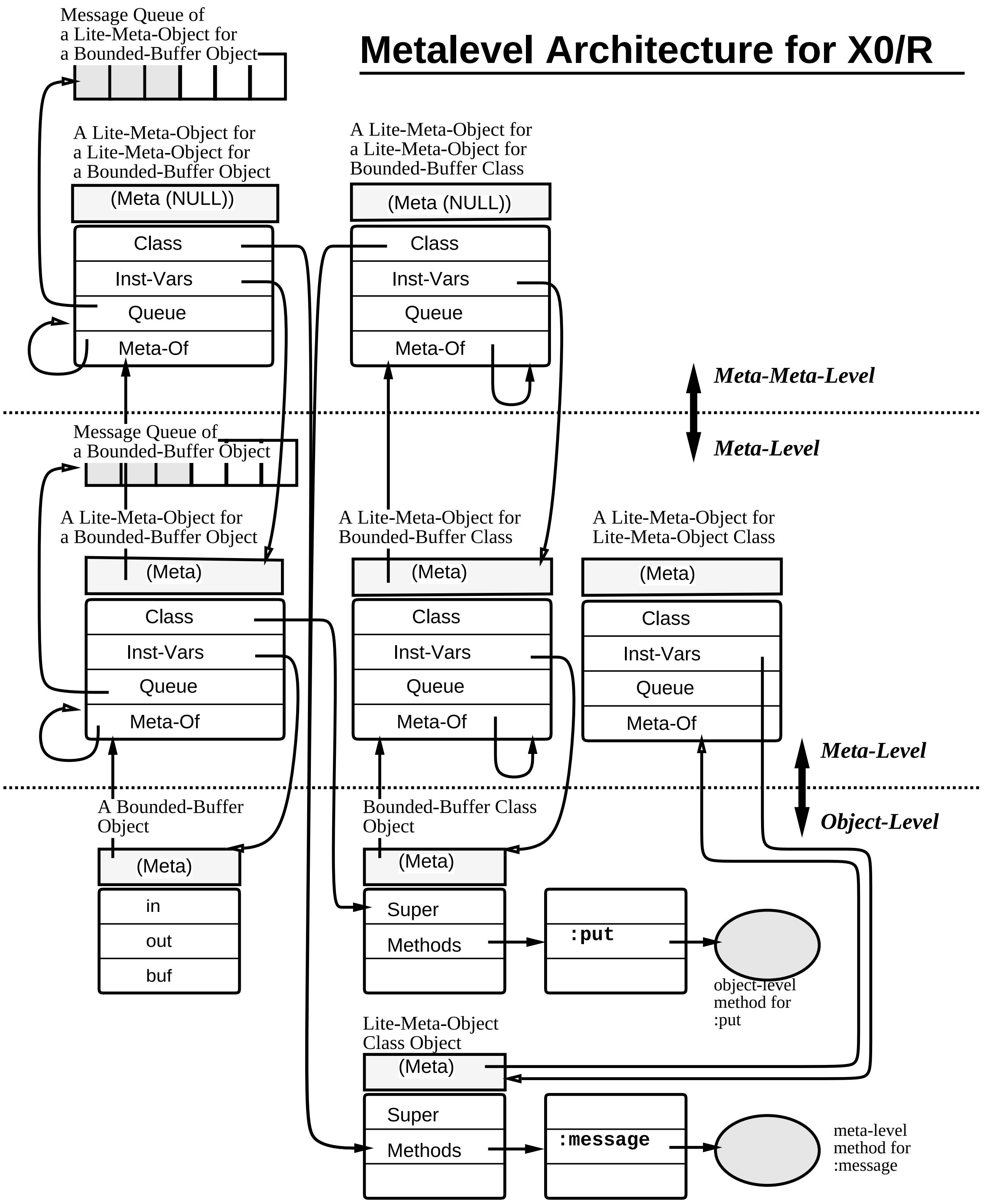
4.1 The Buffer-With-Lock Example

We now give an example of how dynamic progression of degree of reflectivity provided in X0/R would provide a solution to the inheritance anomaly problem while retaining high efficiency. We will inherit from the `Bounded-Buffer` class, adding the `:lock` and `:unlock` methods, in order to create the `Buffer-With-Lock` class. An object, upon accepting the `:lock` message, will be 'locked', i.e., will suspend the reception of further messages until it receives and accepts the `:unlock` message.

As we had noted earlier, `:lock` and `:unlock` are examples of trace-only sensitive methods that cannot be effectively inherited with object-level programming. By employing meta-level operations, however, one would be able to inherit all methods defined at the `Bounded-Buffer` class, and furthermore, all methods defined at the subclasses of `Buffer-With-Lock` will not require any special protocols for maintaining the synchronization constraint of the locks. Below outlines a very simple definition of `:lock` and `:unlock`:

```
[define-class Buffer-With-Lock
  [super Bounded-Buffer]
  [instance-variables Saved-Guards]

  [public [:lock]
          (set! Saved-Guards [[meta Self] <== [:get-all-guards]])
          [[meta Self] <== [:set-all-guards! '#f]]
```



Metalevel Architecture for X0/R
Message Queue of a Lite-Meta-Object for a Bounded-Buffer Object
A Lite-Meta-Object for a Lite-Meta-Object for a Bounded-Buffer Object
(Meta (NULL))
Class
Inst-Vars
Queue
Meta-Of
A Lite-Meta-Object for a Lite-Meta-Object for Bounded-Buffer Class
(Meta (NULL))
Class
Inst-Vars
Queue
Meta-Of
Meta-Meta-Level
Meta-Level
Message Queue of a Bounded-Buffer Object
A Lite-Meta-Object for a Bounded-Buffer Object
(Meta)
Class
Inst-Vars
Queue
Meta-Of
A Lite-Meta-Object for Bounded-Buffer Class
(Meta)
Class
Inst-Vars
Queue
Meta-Of
A Lite-Meta-Object for Lite-Meta-Object Class
(Meta)
Class
Inst-Vars
Queue
Meta-Of
Meta-Level
Object-Level
A Bounded-Buffer Object
(Meta)
in
out
buf
Bounded-Buffer Class Object
(Meta)
Super
Methods
:put
object-level method for :put
Lite-Meta-Object Class Object
(Meta)
Super
Methods
:message
meta-level method for :message

Figure 2: Metalevel Architecture for X0/R

```
          [[meta Self] <== [:set-guard! 'unlock '#t]]]

  [public [:unlock] when #f
          [[meta Self] <== [:set-each-guard! Saved-Guards]]]]
```

These definitions, unfortunately, *do not work*. The reason is that we cannot modify the guards of a single object, for the guards are actually associated with the class object, and are shared by multiple instances of the same class. In more precise terms, by default the meta-object is a `Lite-Meta-Object`; then, although it is possible to reify the guards, it is impossible to reflect the guards into meta-level and maintaining the causal connection.

In order to resolve this situation, we define a 'heavier' meta-object, which allow more extensive degree of reflective operations on guards. We inherit from `Lite-Meta-Object` to create the class `Meta-Object`. It adds an instance variable `guardset`, in which guards that correspond to each method are stored. the method `:message` is re-defined to refer to the stored guards if they are present. The method `:get-all-guards` is also re-defined, actually causing change in the way guards are reified. (This indicates that monolithic reification/reflection would not be sufficient in systems with dynamic progression of degree of reflectivity.)

```
[define-class Meta-Object
   [super Lite-Meta-Object]
   [instance-var
      Guardset)]         ;; an associative vector of guards

   [public [:message M]   ;; overrides Lite-Meta-Object
      ;looks at guards in Guardset instead of inherited guards
           ... ]

   [public [:set-guardset! Gs]
      (set! Guardset Gs)]

   [public [:get-all-guards]     ;; overrides Lite-Meta-Object
      (if (nil? Guardset)
         (set! Guardset [Super <== [:get-all-guards]])
      [! Guardset]]

   [public [:set-guard! M G]     ;; overrides Lite-Meta-Object
      (if (nil? Guardset)
         (set! Guardset [Super <== [:get-all-guards]]))
      (assoc-set! Guardset M G)]

   [public [:set-all-guards! G]  ;; overrides Lite-Meta-Object
      ... ]

   [public [:set-each-guard! Gs] ;; overrides Lite-Meta-Object
      ... ]
]
```

4.2 Automation of Dynamic Progression

The above example presented the meta-level solution to the inheritance anomaly. But why do we not assign the `Meta-Object` as a default meta-object for the class `Buffer-With-Lock`? The reason is that message reception/method dispatch of `Lite-Meta-Object` is very efficient due to the default optimization provided by the system, and since not all lockable objects would require locking, we would like to take advantage of the efficiency as much as possible. We therefore dynamically progress the degree reflectivity only when necessary.

Alteration of the meta-objects to a 'heavier' one is done on demand. To accomplish this, the definitions of the `:set-guard` and other side-effecting methods in `Lite-Meta-Object` *coerces* the meta-object of the denoted object from `Lite-Meta-Object` to `Meta-Object`. The coercion takes place only when necessary, when side-effecting reflective operations are performed on the guard (see the `:set-guard!` method example below). This allows us to maintain the efficiency provided by `Lite-Meta-Object` for the majority of `Buffer-With-Lock` objects which might not require locking. In addition, the coercion need to be performed only once; therefore, subsequent reflective operations on the guards (e.g., `:lock`) will require little overhead. Figure 3 illustrates how the meta-object has been exchanged dynamically to a `Meta-Object` for a `Buffer-With-Lock` object.

```
;; methods for Lite-Meta-Object

[local [:coerce-meta-object! C]
   [temporary new-meta]
   (set! new-meta [C <== [:new [den Self]]])
   [new-meta <== [:set-guardset! [Self <== [:get-all-guards]]]]
   (assoc-set! Inst-Vars 'Meta-Of new-meta)]

[public [:set-guard! m g]   ;; set new guard to corresponding message
   [Self <== [:coerce-meta-object! Meta-Object]]
   [[meta [den Self]] <== [:set-guard! m g]]

[public [:set-all-guards! G]  ;; overrides Lite-Meta-Object, similar
   ... ]
...
```

In order to to maintain the 'progressiveness' of degree of reflectivity, the system does not allow arbitrary meta-object to be specified for replacement: the class of the meta-object is currently restricted to be a descendent of the class of the 'lighter' meta-object.

Notice that the object will remain to be an instance of the same class although the class of its meta-object progresses. This allows us to avoid the various complex semantical issues involving dynamic change of class membership.

5 Conclusion and Future Work

The prime objective of our work is to provide maximum computational power through concurrency of objects. At the same time, the system must be flexible and dynamically configurable, as concurrent computational system is highly complex, and must change and evolve to adapt to the requirements of the user. Some ideas that have flourished in the sequential OO world

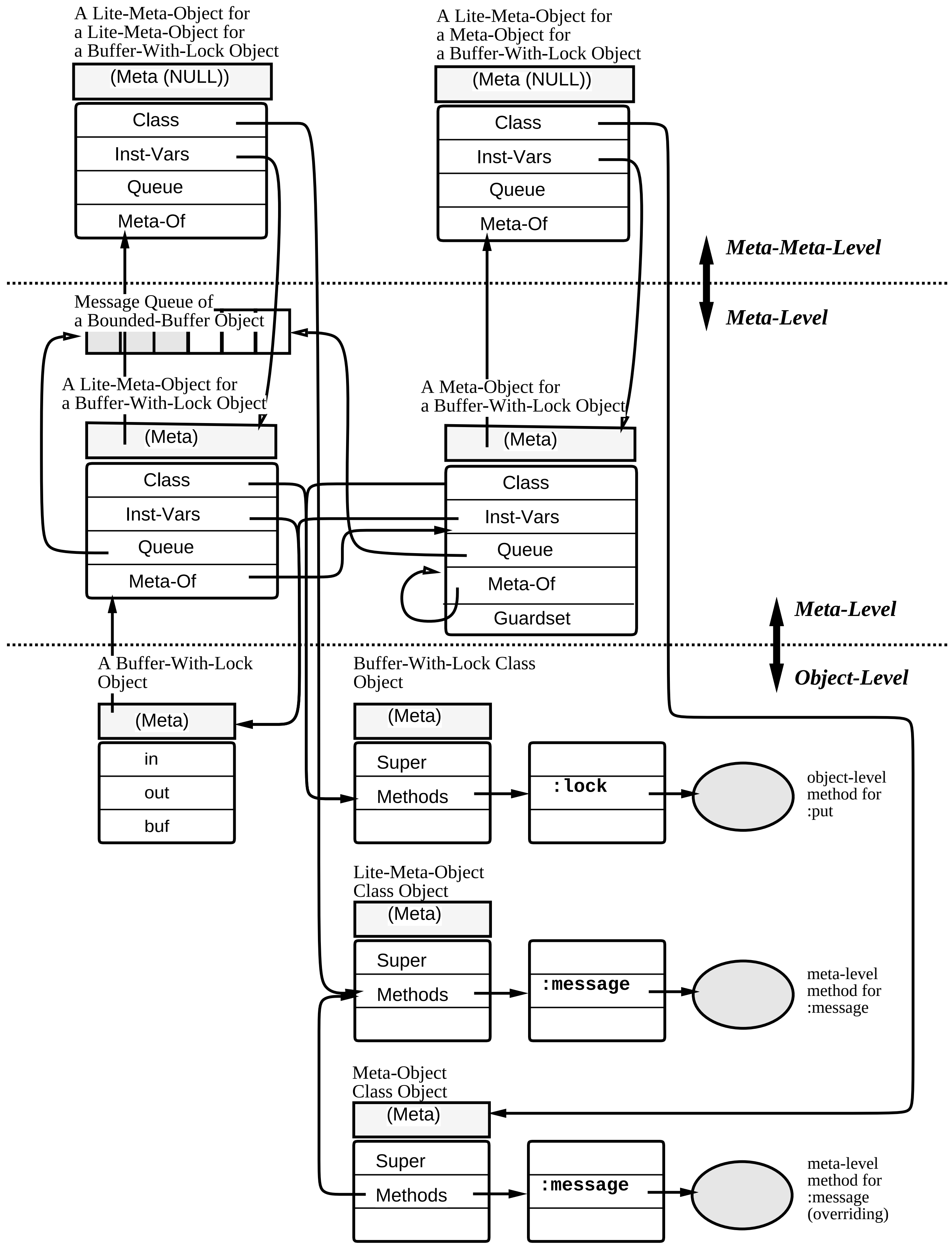


Figure 3: Dynamic Progression of Degree of Reflectivity

such as inheritance have similar objectives; unfortunately, as we have stated, synchronization constraints and inheritance have conflicting characteristics and thus it is difficult to combine them in a clean way. Reflective architecture would provide solutions to this and many other problems facing a OOCP system, both theory and practice: for example there is another research being conducted by the ABCL group in which a group of objects are reflected in the metalevel as a unit[19]. This would serve as a model for more complex systems with different underlying abstractions, such as operating systems.

Dynamic progression could be regarded as somewhat being restrictive compared to arbitrary modification of the behavior of the meta-objects. The argument on our side is that we intend to run *compiled* and *optimized* code. If arbitrary modification is possible, then the computation of the meta-object must be totally reflected in the meta-meta object. Then, the execution of the meta-object would be interpretive, meaning that meta-level operations such as method lookup are executed in an interpretive mode of execution. This clearly conflicts with our interests for achieving high degree of performance with compilation.

Instead, our approach in effect pre-fabricates the necessary meta-objects. We avoid the creation of meta-objects with ad-hoc behavior by utilizing the specialization in the is-a inheritance hierarchy for defining 'heavier' meta-objects. Furthermore, with dynamic progression, we retain the efficiency of compiled code and system optimization; expensive meta-objects allowing more extensive meta-operations are used only when necessary. And even then, the execution of methods is possible with compiled code.

The current status of X0/R is that we have started its implementation on top of Scheme. We are also currently investigating if the progression can be taken further in the cases where the underlying representation of a object would change drastically due to optimization, yet would present the user with an old representation of the object. For example, the optimization program transformation in [13] alters the structure of the object so that an object would have multiple message queues. Although the transformation schema itself is sound, we would like to present the user with a monolithic model of objects and also maintain consistent causal connection between the object and its meta-object. Our solution is to create a 'heavier' meta-object class, `Optimized-Meta-Object`; its definition is sketched below:

```
[define-class Optimized-Meta-Object
   [super Meta-Object]
   [instance-var
      (Queues)             ;; set of message queues for each guard
      (Methodset)          ;; an associative vector of scripts

   [public [:message M]    ;; overrides Meta-Object
      ;does not perform dynamic method lookup, but instead accesses
      ;Methodset
           ... ]
    ...
]
```

Upon receiving the message `[:optimize-transform]`, the object would ask its meta-object to perform the transformation, and switch the meta-object to `Optimized-Meta-Object`. Then, the `Optimized-Meta-Object` would be responsible for creating the illusion of the old object structure, such as a single message queue. With this schema, transformation of an arbitrary object is possible irrespective of its class.

6 Acknowledgments

We would like to thank Takuo Watanabe, Kenny Wakita, Etsuya Shibayama, and Makoto Takeyama for giving us numerous advises and suggestions during the course of our work. We would also like to thank the other members of the ABCL project group for many helpful comments and discussions.

References

[1] P. America. Inheritance and subtyping in a parallel object-oriented language. In *ECOOP '87*, volume 276, pages 234–242. Lecture Notes in Computer Science, Springer Verlag, 1987.

[2] Giuseppe Attardi, Cinzia Bonini, Maria Rosaria Boscotrecase, and Mauro Gaspari. Metalevel programming in clos. In Stephen Cook, editor, *Proceedings of the 1989 ECOOP Conference*, pages 243–256. Cambridge University Press, 1989.

[3] Jean-Piere Briot and Akinori Yonezawa. Inheritance and synchronization in concurrent OOP. In *ECOOP '87*, volume 276, pages 33–40. Lecture Notes in Computer Science, Springer Verlag, 1987.

[4] P. A. Buhr, Glen Ditchfield, and C. R. Zarnke. Adding concurrency to a statically type-safe object-oriented programming language. In *Proceedings of the 1989 Workshop on Object-Based Concurrent Programming*, volume 24(4), pages 18–21. ACM SIGPLAN, ACM, Apr. 1989.

[5] Denis Caromel. A general model for concurrent and distributed object-oriented programming. In *Proceedings of the 1989 Workshop on Object-Based Concurrent Programming*, volume 24(4), pages 102–104. ACM SIGPLAN, ACM, Apr. 1989.

[6] D. Decouchant et al. A synchronization mechanism for typed objects in a distributed system. In *Proceedings of the 1989 Workshop on Object-Based Concurrent Programming*, volume 24(4), pages 105–107. ACM SIGPLAN, ACM, Apr. 1989.

[7] Jim des Rivières. Control-related meta-level facilities in lisp. In Pattie Maes and Daniele Nardi, editors, *Meta-Level Architectures and Reflection*, pages 101–110. North-Holland, 1988.

[8] Jacques Ferber. Computational reflection in class-based object-oriented languages. In *Proceedings of the 1989 OOPSLA Conference*, volume 24, pages 317–326. ACM SIGPLAN, ACM, Oct. 1989.

[9] Brian Foote. Reflective facilities in smalltalk-80. In *Proceedings of the 1989 OOPSLA Conference*, volume 24, pages 327–335. ACM SIGPLAN, ACM, Oct. 1989.

[10] Dennis G. Kafura and Keung Hae Lee. Inheritance in actor based concurrent object-oriented languages. In *ECOOP '89*, pages 131–145. Cambridge University Press, 1989.

[11] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of the 1987 OOPSLA Conference*, volume 22, pages 147–155. ACM SIGPLAN, ACM, Oct. 1987.

[12] Satoshi Matsuoka and Satoru Kawai. Using tuple space communication in distributed object-oriented languages. In *Proceedings of the 1988 OOPSLA Conference*, volume 23(11), pages 276–283. ACM SIGPLAN, ACM, Sep. 1988.

[13] Satoshi Matsuoka, Ken Wakita, and Akinori Yonezawa. Synchronization constraints with inheritance: What is not possible — so what is? Technical Report 10, Dept. of Information Science, The Univ. of Tokyo, April 1990.

[14] O. M. Nierstrasz. Active objects in Hybrid. In *Proceedings of the 1987 OOPSLA Conference*, volume 22, pages 243–253. ACM SIGPLAN, ACM, Oct. 1987.

[15] M. Papathomas. Concurrency issues in object-oriented programming languages. In D. Tsichritzis, editor, *Object Oriented Development*, chapter 12, pages 207–245. Universite de Geneve, 1989.

[16] Chris Tomlinson and Vineet Singh. Inheritance and synchronization with enabled-sets. In *Proceedings of the 1989 OOPSLA Conference*, volume 24(10), pages 103–112. ACM SIGPLAN, ACM, Oct. 1989.

[17] Jan van den Bos and Chris Laffra. PROCOL a parallel object language with protocols. In *Proceedings of the 1989 OOPSLA Conference*, volume 24(10), pages 95–102. ACM SIGPLAN, ACM, Oct. 1989.

[18] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings of the 1988 OOPSLA Conference*, volume 23, pages 306–315. ACM SIGPLAN, Sep. 1988.

[19] Takuo Watanabe and Akinori Yonezawa. An actor-based metalevel architecture for group-wide reflection. Submitted to the OOPSLA/ECOOP-90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Languages, Ottawa, Canada, aug 1990.

[20] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A reflective architecture for an object-oriented distributed operating system. In Stephen Cook, editor, *Proceedings of the 1989 ECOOP Conference*, pages 89–106. Cambridge University Press, 1989.

[21] Akinori Yonezawa, editor. *ABCL An Object-Oriented Concurrent System.* Computer Systems Series. MIT Press, Jan. 1990.

[22] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *Proceedings of the 1986 OOPSLA Conference*, volume 21, pages 258–268. ACM SIGPLAN, Sep. 1986.