Avoiding Confusion in Metacircularity: The Meta-Helix*

Shigeru Chiba Gregor Kiczales John Lamping

Xerox Palo Alto Research Center
chiba|gregor|lamping@parc.xerox.com

Keywords: metaobject protocol, reflection, metacircular, CLOS

Abstract

A system with a metaobject protocol (MOP) allows programmers to extend it and then use the extended system as naturally as the original non-extended one. Such metaobject protocols often use a metacircular architecture to make the extensions easier to write. Unfortunately, this use of metacircularity can lead to problems stemming from a conflation of the extended and non-extended functionalities. We present a new architecture, called the meta-helix, that preserves the advantages of metacircularity but also addresses the problems with conflation.

1 Introduction

A system with a metaobject protocol (MOP) allows programmers to extend it, and then use the extended system as naturally as the original non-extended system. Metaobject protocols have been designed and implemented for programming languages [KdRB91, Mae87, WY88, MWY91], window systems [Rao91], and operating systems [Yok92, YTR+87, MA90, Mae96].

A common element of metaobject protocol design is the use of metacircularity to allow extensions to be implemented in terms of the original non-extended functionality. This can greatly simplify the implementation of a number of extensions. Unfortunately, it can also lead to a conflation of the extended and non-extended functionalities, which in turns complicates both the code that uses the extension and the code that implements the extension.

In this paper, we present an analysis of this problem and a general architectural solution which we call the meta-helix. The intuition behind the meta-helix is that even though the extended and non-extended languages may be very similar, the fact that one is used to implement the other makes it important to reify them as distinct entities. The meta-helix makes it possible to preserve what is good about metacircular MOPs while

*Appeared in Proc. of the 2nd Int'l Symp. on Object Technologies for Advanced Software (ISOTAS), LNCS 1049, Springer, pp.157–172, 1996.

at the same time resolving the problem we have identified. We show how the meta-helix can be implemented in both compile-time and runtime MOPs.

2 Using Metaobject Protocol Based Extensions

In this section we show an example of using a MOP-based extension, specifically an extension to the Common Lisp Object System (CLOS) [BDG+88], which the CLOS MOP [KdRB91] enables.[1] In this extension the object system's original notion of slots is augmented to support slots with an access history. That is, slots which store not only their current value, but also a history of all their reads and writes.

Following is an example of a program that uses the extension. It defines a class `point`, with two slots `x` and `y`. The third line of the definition specifies that the class `point` uses the extended version of class that supports slots with history.

```
(defclass point ()
     (x y)
  (:metaclass history-class))
```

All the accesses to the slots of instances of `point` are recorded, and the access log is available via the function `slot-history`. For example, the following program fragment builds an instance of `point`, accesses some of its slots, and then prints out the whole history:

`(setq p1 (make-instance 'point))`	make a point, `p1`
`(set-slot p1 'x 3)`	set the `x` slot to 3
`(get-slot p1 'x)`	read slot `x`
`(set-slot p1 'y 5)`	set the `y` slot to 5
`(print (slot-history p1))`	this last expression
`  ((set x 3) (get x) (set y 5))`	shows the whole sequence of events

3 Implementing Extensions Using Metacircularity

The CLOS MOP is designed to allow meta-programs that implement extensions to themselves use the full power of non-extended CLOS. This property—referred to as metacircularity—makes it much easier to implement many extensions, since they can use the powerful facilities of the system being extended, rather than having to rely on lower-level functionality.

In this case, slots with access history can be implemented using the underlying non-extended slots. The way this works is simply for instances of the class `point` to actually have three slots, the two visible slots `x` and `y` as well as a third, "hidden" slot `history`

[1] To make it easier for non CLOS programmers to read the paper, we use a slightly simplified version of the CLOS syntax and the CLOS MOP. The simplification is carefully chosen not to affect the argument the paper presents.

for storing the history. The code for implementing the slots with history extension is as follows (note that this code has an important bug, which will be addressed shortly):

```
(defclass history-class (class) ())

; Every history class should have an extra slot,
; to store the history
;
(defmethod compute-slots ((c history-class))
  (cons 'history (call-next-method)))

; Specialize get-slot so that it updates the history before
; using call-next-method to invoke the non-extended
; functionality.
;
(defmethod do-get-slot
           ((c history-class) object slot-name)
  (set-slot object
            'history
            `((get ,slot-name)
              ,@(get-slot object 'history))[2])
  (call-next-method))

(defmethod do-set-slot ...)
```

The simplicity of this code is due in large part to its use of the CLOS MOP's metacircularity. The implementor of the slots with history extension has the full power of CLOS available in which to implement the extension, and that makes the code simpler.

4 Implementation Level Conflation

Unfortunately, this kind of metacircularity can also lead to confusion between the extended and non-extended functionality. To see how, it is first important to notice that with this extension in place, there are *two concepts* of class, object and slot in the world. There is the original, non-extended concepts of class, object and slot, in which the history of slots is not recorded. Then there is the extended language, with its own concept of class, object and slot.

Moreover, the parallel concepts of class, object and slot exist not only side-by-side—in that some entirely unrelated class like `astronaut` might not be using `history-class`—but also in an *implemented-by* relationship. Extended slots that record their history are implemented in terms of non-extended slots that don't.

The problem in which we are interested arises when a MOP fails to appropriately reify the distinction between the extended and non-extended functionalities. We call this *implementation level conflation* because the implemented and implementing levels are conflated into a single structure.

The way this appears in the example program is illustrated in Figure 1. There is only one point object, `p1` and only one class metaobject, `point`; there is no distinction

[2]Readers who are unfamiliar with the Lisp backquote facility should read this expression as: `(cons (list 'get slot-name) (get-slot object 'history))`.

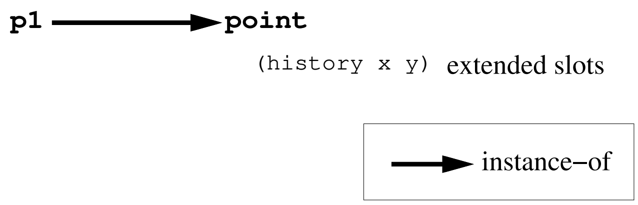

Figure 1: Implementation level conflation in the CLOS MOP. The implemented and implementing objects are conflated into a single object (i.e. `p1`). The implemented and implementing classes are conflated into a single class metaobject (i.e. `point`).

between a point object with the extended slots functionality and an implementing object without the extended slots. This implementation level conflation results in confusion for both users and implementors of extensions.

Problems for Users of the Extension

Implementation level conflation can cause problems for users of the extension, such as for another part of the example program. Consider, for example, code that uses one of the MOP's introspective facilities to ask what slots a class has:

```
(print (class-slots (find-class 'point)))

  (history x y)
```

Notice that the result includes the `history` slot. But this program, which is a *user* of the extension, wants to operate in terms of implement*ed* classes and objects, not implement*ing* classes and objects. So, for this result to include the `history` slot is entirely inappropriate, it shouldn't know it even exists. This problem particularly shows up when using browsers and debuggers that rely on the introspective part of the MOP to work. Exposing this detail of how the slots with history extension is implemented can leave programmers confused, or worse yet, can tempt them to rely on this implementation detail in ways that they shouldn't.

Problems for the Implementor of the Extension

Implementation level conflation can also affect the implementor of the extension. A careful reading of the method for `do-get-slot` shown in Section 3 above provides an example of this.

This code has a bug which manifests itself as infinite recursion, but is better understood as resulting from implementation-level conflation. Operationally, the bug is that the body of the method, as part of updating the history, must read the `history` slot, and so calls `get-slot` recursively, which runs this method, which starts to update the history, which reads the history slot, and so on ad infinitum.

The standard solution to this problem is to introduce a special purpose test that prevents the infinite recursion.[3] This is often called "making sure the tower grounds out" [dRS84, dR90]. This solution, while effective, seems ad hoc, can be difficult to reason about, and is not effective in general.

This problem can be better described as having to do with implementation level conflation. As shown below, the issue is that there are two concepts of slot in play: the implemented history slots, and the implementing slots without history[Dix91]. But, because of the conflation, there is only one thing `get-slot` (and `set-slot`) of an object can possibly refer to, which in this case turns out to be slots with history.

slots with history

```
(defmethod get-slot ((c history-class) object slot-name)
  (set-slot object
            `history
            (cons `get ,slot-name)
                  (get-slot object 'history)))
  (call-next-method))
```

ordinary slots

If, somehow, the references to the concept of slots in the body of the method could be to non-extended slots, the ad-hoc solution to the infinite recursion could be avoided.[4]

4.1 The Importance of an N-to-N Correspondence

Fundamentally, there needs to be some way for instances of the class `point` to be viewed in terms of either the implemented functionality or the implementing functionality, not a conflation of both. As discussed above, the user of the extension wants a view in terms of the implemented functionality. The implementor of the extension wants to be able to take one view or the other at different times. Taking an additional step back, we can say that if there are n distinct important views of an object—or any other structure—there needs to be n distinct handles to it.

[3] The revised code ends up looking something like:

```
(defmethod do-get-slot
           ((c history-class) object slot-name)
  (unless (eq slot-name 'history)
    (set-slot object
              'history
              `((get ,slot-name)
                ,@(get-slot object 'history))))
  (call-next-method))
```

[4] The use of C++'s qualified member function call, which makes it possible to directly invoke a method supplied by a specific super-class, could be used to solve this particular problem. Unfortunately, it does not the address the more general problem of implementation level conflation.

5 Solutions

In this section, we propose a solution to the problem of implementation level conflation in metacircular systems, based on a concept that we call the meta-helix. Rather than presenting the solution immediately, we first present two earlier solutions that, in different ways, fail to meet our needs. These solutions serve to further flesh out the criteria which the more general solution should meet.

5.1 A First Inadequate Solution

One possible solution to this problem involves implementing the extension in a different way, specifically by storing the slot histories in the class metaobject rather than directly in the objects themselves. The following meta-level code implements history classes this way:

```
(defclass history-class (class)
     (histories))               ; Slot histories will be stored
                                ; in the class metaobject.

(defmethod do-get-slot
            ((c history-class) object slot-name)
  (let ((history-storage
           (slot-history-storage c object)))
    (setf (cdr history-storage)
           `((get ,slot-name)
             ,@(cdr history-storage)))
    (call-next-method)))

(defmethod do-set-slot ...)

; This support function maintains a correspondence to each
; individual object's slot history.
;
(defun slot-history-storage (c object)
  (let* ((entries (get-slot c 'histories))
          (entry (assoc entries object)))
    (if entry
        entry
        (let ((new-entry (list object)))
          (set-slot c
                    'histories
                    (cons new-entry entries))
          new-entry))))
```

This solution solves the specific problems mentioned in Section 4, but it has several significant problems of its own, which we would like a more general solution to avoid. These include:

- First, it loses many of the advantages of metacircularity. This solution must manually implement the mapping from individual objects to associated data (i.e. `slot-history-storage`), even though that basic functionality is already present in CLOS.

- Second, it is difficult to optimize automatically. While this paper does not address the performance issue at length, it is worth noting that the extra hand-coded mapping from an object to its associated storage won't be optimized in the way that normal slot access is optimized, and so slots with history will be unduly slow to access.

5.2 A Second Inadequate Solution

A second possible solution comes from the Tiny CLOS MOP, which is an extremely simplified version of CLOS with a MOP that we developed for research purposes. One goal of the Tiny CLOS MOP design was to address the problem of implementation level conflation in a way that does not have the problems associated with the first solution.

To do this, the Tiny CLOS MOP provides two different abstractions for per instance storage: slots and fields. Fields are a lower-level abstraction used to implement slots; they have a more primitive naming mechanism in terms of indices. Base-level Tiny CLOS programs never know that fields exist, they only operate in terms of slots. Meta-level programs that extend slot functionality do so by implementing slots in terms of fields.

In the specific example of the history class, the extension works by allocating an extra *field* for each object. So, for example, `point` objects have two slots `x` and `y`; but they have three fields, for holding the `x` and `y` slots and the slot access history.

The implementation of the slot access history extension in Tiny CLOS looks like:[5]

```
(defclass history-class (class)
    (history-index))          ;the index of the field that will
                              ;store the history for instances
                              ;of the class

; Allocate an extra field for the slot access history and
; remember its index in the class.
;
(defmethod compute-fields ((c history-class))
  (set-slot c 'history-index (allocate-field c))
  (call-next-method))

(defmethod do-get-slot
           ((c history-class) object slot-name)
  (let ((index (get-slot c 'history-index)))
    (set-field object
               index
               `((get ,slot-name)
                 ,@(get-field object index)))
    (call-next-method)))

(defmethod do-set-slot ...)
```

Again, this solution solves the specific problems mentioned in Section 4. Implementation level conflation is avoided because there are distinct abstractions (views) for slots

[5] The actual syntax of Tiny CLOS is more primitive, but to make the paper easier to read, we present this code in a syntax similar to the CLOS syntax we have been using. Again, we have been careful not to affect the validity of the paper in doing so.

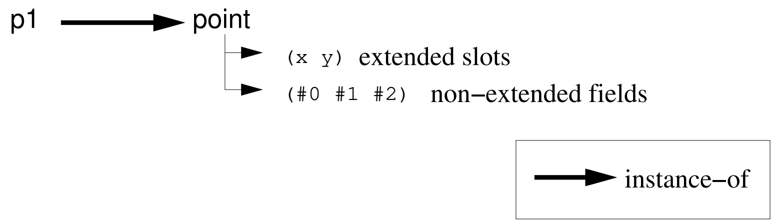

Figure 2: In Tiny CLOS there is only a single object, and a single class metaobject, but the class metaobject splits, to some extent, into two views, one in terms of slots and the other in terms of fields.

and fields. The slot abstraction is used for the implemented functionality and the field abstraction is used for the implementing functionality. Users of the extension can ask for the slots of a class and not get the history. The implementation of the extension can ask for either the slots or the fields, and can do access in terms of either the slots or the fields.

Unfortunately, this solution has significant problems of its own which make it unsuitable as a general solution.

- As with the previous solution, the Tiny CLOS approach loses many of the advantages of metacircularity. Note that this code uses `set-field` and `get-field` to access the history, and the fact that the extension must itself keep track, on a per-class basis, of the location of the field for the slot access history. Because this part of the Tiny CLOS MOP is not metacircular in the way the CLOS MOP is, the extension programmer must learn a new abstraction (i.e. fields) and programming in that lower-level abstraction can be more complex.

- The second problem is that this solution is only effective in the presence of a single extension to slot functionality. If, for example, someone wanted an additional extension (i.e. to store the objects in a persistent database [Pae90]) there would still be conflation. This is because in such a situation there needs to be (at least) 3 views. The view of persistent objects with a slot access history, built on top of the view of persistent objects, built on top of ordinary objects. But the Tiny CLOS MOP provides only two levels, so there will still be some conflation. To avoid implementation level conflation in n levels of implementation, the MOP must provide support for n different views.

5.3 The Meta-Helix

The common idea underlying the two unsatisfactory solutions is to distinguish the implemented and implementing functionality by using a different "handle" for each. In the first proposed solution the objects are the handle to the implemented functionality, the class metaobject is the handle to the implementing functionality. Figure 2 shows how

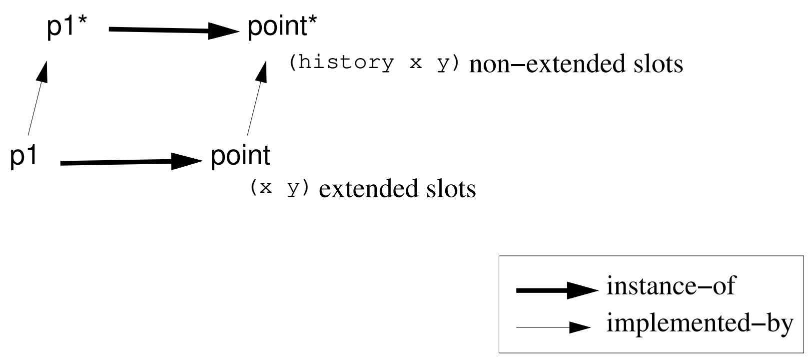

Figure 3: The implemented and implementing levels are distinctly reified in our approach.

Tiny CLOS, while providing only a single class metaobject, does provide two views of that class, one in terms of slots and one in terms of fields.

But the problem with both of these solutions is that the use of the two distinct mechanisms destroys the benefits of metacircularity. The idea of our proposed solution is to retain the benefits of metacircularity, but address its problems by reifying the implemented-by relationship. In addition to the usual `instance-of` and `subclass-of` relationships, we introduce an `implemented-by` relationship to capture the relationship between implemented and implementing objects and class metaobjects.

For example, in the case of the `history-class` extension, our solution has two class metaobjects, `point` and `point*`, to represent different implementation levels (Figure 3). The class `point` corresponds to the class in the extended language, which keeps slot access histories, while the class `point*` corresponds to the class in the non-extended language, which is used to implement the extended one.

The crucial difference between Figure 3 and Figure 2 is in the use of the implemented-by relationship. That is what allows implemented and implementing classes to be kept distinct. The use of the implemented-by relation avoids conflation. It recognizes that, however similar they may be, the implemented and implementing classes describe distinct objects which must not be conflated. Thus, the `history` slot of `point*` is not a slot of `point`.

Our choice of the name meta-helix for this architecture is best seen when thinking in terms of the relation between interfaces that the different solutions use. As shown in Figure 4, in the pure metacircular approach, the implementation loops directly back onto itself—leading to conflation. In the Tiny CLOS approach, the implementation maps between two distinct functionalities—leading to added complexity. In the meta-helical approach, the implementation spirals between two distinct interfaces of (nearly) identical functionality—preserving what is good about the metacircular approach, while still reifying the distinction that prevents conflation of implementation levels.

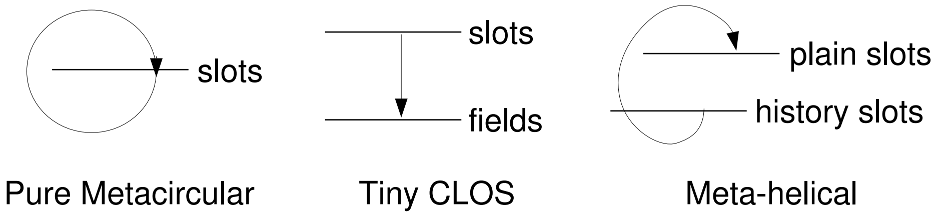

Figure 4: The implementation relation between interfaces.

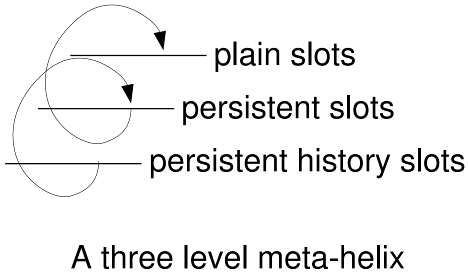

Figure 5: The Meta-Helix Architecture Supports n Implementation Levels.

While distinguishing implementation levels, the meta-helix preserves the benefits of metacircularity, because the implementing classes are free to use the full power of the language being extended. So, for example, unlike the Tiny CLOS solution, programmers do not need a different abstraction such as fields to implement an extended concept of slots.

Similarly, the meta-helix works when there are more than two implementation levels. So, for example, in the case of the persistent history class mentioned above, we can create the three levels that are needed to maintain separate views, and relate them using implemented-by relationships. This is shown in Figure 5, which illustrates the helical nature of this architecture.

5.4 Proper Use of the Meta-Helix

There is a seductive mistake that can be when implementing an extension using the meta-helix. The implemented class is often very similar to the implementing class, as in this example. This makes it easy to imagine that the implemented class is an extension of the implementing class, and should be implemented as a subclass of the implementing class (i.e. `point` would be a subclass of `point*`). But it is important not to do this, because it would once again be implementation level conflation. Some properties of `point*`, like having a `history` slot, must *not* become properties of `point`. The actual relationship between `point` and `point*` is implemented-by, not extension-of.

6 Implementing the Meta-Helix

Since the meta-helix applies to several different kinds of MOP, we present meta-helix versions of several styles of MOP. All of these will be class-based MOPs, rather than object-based MOPs. That is, classes will have metaobjects, but objects won't (except via their classes). We will thus have a different class metaobject for each layer of an implementation hierarchy. Each will know what class metaobject to use to implement it.

6.1 A Compile-Time MOP Based on the Meta-Helix

We first present a compile-time MOP based on the meta-helix. In a compile-time MOP, the meta-level code runs only at compile time. The meta-level code controls the compilation of the program, and thereby indirectly controls its runtime behavior [Rod92, LKRR92, Chi95].

A compile-time MOP controls compilation from its source language to its target language. For a compile time MOP based on the meta-helix, the target language is simply the non-extended language. The compile-time MOP we present will control translation from extended CLOS to non-extended CLOS.[6]

The MOP controls the translation of programs that use extensions, such as slots with history, into programs in non-extended CLOS. The translated programs are then compiled by ordinary CLOS compilers. Since this MOP is a translator from extended CLOS to CLOS, the translated program will be identical to the source program, except where it makes use of extended features. The translator thus behaves as the identity function by default. This default behavior is extended to handle the translation of extensions.

To implement the `history-class` extension shown in Section 2, the programmer specializes the translation of classes and slot access primitives for `history-class` classes. For example, the definition of the class `point` in the source program:

```
(defclass point ()
    (x y)
  (:metaclass history-class))
```

is translated into a definition of the class `point*` and a method for `slot-history`:

```
(defclass point* ()
    (history x y))

(defmethod slot-history ((object point*))
  (get-slot object 'history))
```

Note that the definition of `point*` is in non-extended CLOS. Instances of that class have three non-extended slots `history`, `x` and `y`. All occurences of the class `point`

[6] The actual meta-helical compile-time MOP we have developed—described in [Chi95]—translates from extended C++ to non-extended C++; but for simplicity in this paper we continue with a presentation in terms of a CLOS-like syntax. Again, we have been careful not to affect the validity of the paper in doing so.

in the source program are replaced with `point*` during the translation. The objects created as instances of `point` are thus implemented by instances of `point*`.

In compile-time MOPs based on the meta-helix, the implemented-by relationship in Figure 3 is thus manifested as a compile-time translation from the implemented class to the implementing class. Thus, the class `point` is implemented by the class `point*`.

The meta-level code does the translation. That is, the meta-level code that implements `history-class` classes is responsible for creating a new implementing class for each `history-class` class, `point*` in this case, and for translating all other aspects of the implemented class into the appropriate operations on the implemented class. The MOP guarantees that all aspects of an implemented class will go through the translation process, so that there will be no conflation of levels.

Thus, all occurences of the slot access expression `(get-slot` *object slot-name*`)` on `point` objects will also go through a translation process, resulting in the CLOS expression:

```
(progn (set-slot object
                 'history
                 (cons `(get slot-name)
                       (get-slot object 'history)))
       (get-slot object slot-name))
```

Note that the translated expression will be executed on instances of the class `point*`, since all occurrences of `point` are replaced with `point*`. The `get-slot` and `set-slot` primitives in the translated expression will thus operate on `point*` objects, so their behavior will be the default one. Thus the infinite recursion on `get-slot` that showed up in Section 3 is not an issue here.[7]

6.2 A Runtime MOP based on the Meta-Helix

The meta-helix is applicable to run-time MOPs as well as to compile-time MOPs. In this section, we discuss how to adapt one run-time MOP, the CLOS MOP, to be meta-helical.

In compile-time meta-helical MOPs, the translation process reduces all constructs in the extensions to non-extended constructs, so that the final code which is executed at run-time contains only non-extended constructs. A call for creation of an extended object will have been translated—possibly through several levels—to a call for creation of a non-extended object that implements it. In a literal sense, only the non-extended objects exist at run-time.

In a run-time meta-helical MOP, on the other hand, the implementation process must be carried out at run-time, so there will be a run-time object for each level of implementation. The implemented-by relation becomes manifested as an `implemented-by` field in every extended object, which gives the object one level down that implements it.

Just as in a compile-time meta-helical MOP, the metaobject for an extended class is responsible for defining the class that will implement it. But now, both classes will exist at run-time. The metaclass for `point` will create a class equivalent to the definition:

[7] Of course, if `history-class` had been erroneously implemented in terms of itself, that is if `history-class` classes translated into `history-class` classes, then there would be in infinite recursion of translation. The meta-helix doesn't eliminate real implementation circularities, it just avoids extraneous ones.

```
(defclass point* ()
    (history x y))
```

At runtime, the class `point` delegates most of its slot access work to the class `point*`. For example, when an instance of `point` is created, the class `point` asks the class `point*` (its implementing class) to create a `point*` instance. All the `point` object contains is a tag saying that it is a `point` and an implemented-by field that holds the `point*` object.

The slot access primitive `get-slot` is then specialized by the class metaobject for `point`. Its implementation is:

```
(defmethod do-get-slot
           ((c history-class) object slot-name)
  (let ((implemented-by (implemented-by object)))
    (set-slot implemented-by
              'history
              `((get-slot ,slot-name)
                ,@(get-slot implemented-by
                            'history)))
    (get-slot implemented-by slot-name)))
```

The meta-level generic function is passed the object of the extended class. The method first gets the implementing object for that object (i.e. an instance of `point*`) and then operates on the implementing object to carry out the actual work of implementation. Thus, when it invokes `get-slot` internally, it is invoking it on a `point*` object. The levels are not conflated. This delegation keeps the levels clearly separated.

7 Related Work

There are numerous metacircular MOPs with architectures that differ in important ways from that of the CLOS MOP. The Smalltalk-80 MOP [GR83] distinguishes classes and metaclasses, the 3-KRS MOP [Mae87] creates distinct metaobjects for individual objects, and the ObjVlisp MOP [Coi87] takes a uniform model of metaclasses. But these MOPs also suffer from the problem of implementation level conflation, because none of those differences reify the implemented-by relation in the way the meta-helix does.

The AL-1/D MOP [OIT92] provides multiple views in that objects are controlled by multiple metaobjects each of which represents different aspects of implementation, such as operational behavior and resource allocation. But the AL-1/D MOP is not metacircular, and so the issue of separation between the implemented view and the implementing view does not arise in the same way. The focus in AL-1/D is on dividing the implementing view into different parts.

Work on subject-oriented programming [HO93, SZ89] is targeted at one of the underlying problems we have faced—how to have multipled clearly distinguished views of an object. But our focus here is not on the general issues of subjectivity, but rather on the more specific problem of implementation-level conflation in metacircular MOPS. Our solution is informed by that work, but is more specific to our needs.

8 Conclusion

We have developed a new analysis of a problem that arises in existing metacircular metaobject protocols. This analysis, in terms of the notion of implementation level conflation, shows how pure metacircularity causes confusion when there aren't clearly distinguished views of the implemented and implementing functionality.

A new architectural approach, called the meta-helix, retains the benefits of metacircularity, but address the traditional problems. It does this by reifying the implemented-by relation, so that even if the implementing language is the same as the implemented language, the two levels are reified distinctly. In a meta-helical MOP, the implementing code can be written in the non-extended language—this preserves the benefits of metacircularity. A clear separation between implemented and implementing objects and classes prevents the traditional problems that stem from implementation level conflation.

Acknowledgements

We would like to thank the members of the Embedded Computation Area at Xerox PARC who commented on earlier drafts of this paper. We would also like to thank the anonymous reviewers for suggestions that improved the quality of the presentation.

References

[BDG+88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common lisp object system specification. *Sigplan Notices*, 23(Special Issue), September 1988.

[Chi95] Shigeru Chiba. A metaobject protocol for C++. In *OOPSLA '95 Conference Proceedings Object-Oriented Programming Systems, Languages, and Applications (to appear)*, October 1995.

[Coi87] Pierre Cointe. Metaclasses are first class: The ObjVlisp model. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Orlando, FL*, pages 156–167, 1987.

[Dix91] Michael Dixon. *Embedded Computation and the Semantics of Programs*. PhD thesis, Stanford University, 1991.

[dR90] Jim des Rivières. The secret tower of CLOS. In *Informal Proceedings of ECOOP/OOPSLA '90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1990.

[dRS84] Jim des Rivières and Brian Cantwell Smith. The implementation of procedurally reflective languages. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 331–347. ACM, 1984.

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

[HO93] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In Andreas Paepcke, editor, *OOPSLA '93 Conference Proceedings Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428. ACM/SIGPLAN, ACM Press, October 1993. Volume 28, Number 10.

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

[LKRR92] John Lamping, Gregor Kiczales, Luis H. Rodriguez Jr., and Erik Ruf. An architecture for an open compiler. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992. Also to appear in forthcoming PARC Technical Report.

[MA90] Dylan McNamee and Katherine Armstrong. Extending the mach external pager interface to allow user-level page replacement policies. Technical Report UWCSE 90-09-05, University of Washington, September 1990.

[Mae87] P. Maes. Concepts and experiments in computational reflection. In *OOPSLA '87 Conference Proceedings, Sigplan Notices* **22***(12)*, pages 147–155. ACM, Dec 1987.

[Mae96] Chris Maeda. A metaobject protocol for file systems. In *Proceedings of ISOTAS'96 (International Symposium on Advanced Technologies for Object Software)*, March 1996.

[MWY91] Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *European Conference on Object Oriented Programming*, pages 231–250, 1991.

[OIT92] Hideaki Okamura, Yutaka Ishikawa, and Mario Tokoro. Al-1/d: A distributed programming system with multi-model reflection framework. In *Proceedings of the International Workshop on New Models for Software Architecture '92; Reflection and Meta-Level Architecture*, pages 36–47, 1992.

[Pae90] Andreas Paepcke. PCLOS: Stress testing CLOS Experiencing the metaobject protocol. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications and the European Conference on Object-Oriented Programming*, pages 194–211, 1990.

[Rao91] Ramana Rao. Implementational reflection in silica. In *European Conference on Object-Oriented Programming*, pages 251–266, 1991.

[Rod92] Luis H. Rodriguez Jr. Towards a better understanding of compile-time mops for parallelizing compilers. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992. Also to appear in forthcoming PARC Technical Report.

[SZ89] Lynn Andrea Stein and Stanley B. Zdonik. Clovers: The dynamic behavior of types and instances. Technical Report CS-89-42, Brown University, 1989.

[WY88] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In *Object Oriented Programming, Systems, Languages, and Applications Conference Proceedings*, pages 306–315, 1988.

[Yok92] Yasuhiko Yokote. The apertos reflective operating system: The concept and its implementation. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 414–434, October 1992.

[YTR+87] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th Symposium on Operating Systems Principles*, 1987.