Experience with CommonLoops

James Kempf
Warren Harris
Roy D'Souza
Alan Snyder

Hewlett-Packard Laboratories
1501 Page Mill Rd., Palo Alto, CA, 94304

Abstract

CommonLoops is an object-oriented language embedded in Common Lisp. It is one of two such languages selected as starting points for the Common Lisp Object System (CLOS) which is currently being designed as a standard object-oriented extension to Common Lisp. This paper reports on experiences using the existing Portable CommonLoops (PCL) implementation of CommonLoops. The paper is divided into two parts: a report on the development of a window system application using the CommonLoops programming language, and a description of the implementation of another object-oriented language (CommonObjects) on top of the CommonLoops metaclass kernel, paralleling the two aspects of CommonLoops: the programming language and the metaclass kernel. Usage of the novel features in CommonLoops is measured quantitatively, and performance figures comparing CommonLoops, CommonObjects on CommonLoops, and the native Lisp implementation of CommonObjects are presented. The paper concludes with a discussion about the importance of quantitative assessment for programming language development.

1. Introduction

CommonLoops [Bobrow86] is an object-oriented extension to the Common Lisp programming language [Steele84], and is one of two such languages[1] selected as starting points for the Common Lisp Object System (CLOS) [ANSI87], a standard object-oriented Lisp extension currently under discussion. Since it is probable that the CLOS (and thus the features of CommonLoops incorporated into it) will see widespread use, both for applications prototyping and for production code, it seems prudent that the novel features of CommonLoops be assessed for their utility in object-oriented programming. A similar, though more formalized, review procedure was followed with Smalltalk-80 when it was first released [McCullough83] [Falcone83] [Ballard83].

The novel features of CommonLoops discussed in [Bobrow86] include the following:

- It is the first object-oriented language or object-oriented language extension to appear in a highly portable, public domain implementation[2]. This implementation has proven to be an invaluable contribution to education and research in object-oriented programming,
- The partial integration of the Common Lisp type system with CommonLoops classes which are themselves first class objects. This allows method dispatching on certain Common Lisp data objects having standard Common Lisp types, as well as on objects which are instances of CommonLoops classes,
- The use of generic function syntax for method dispatching rather than a specific method invocation operator[3],
- The ability to define methods that discriminate on more than just the first argument. These methods are called "multimethods," as opposed to "classical methods," which dispatch only on the first argument,

1. The other is New Flavors [Moon86].

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commerical advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1987 ACM 0-89791-247-0/87/0010-0214 $1.50

2. "Highly portable" means that a Common Lisp programmer with a complete implementation of Common Lisp can generally bring up CommonLoops in a morning. Object-oriented languages such as C++ [Stroustrup86] and Objective-C [Cox86] are "moderately portable," since their compilers and preprocessors can be ported, but porting requires considerable time and effort.
3. Another object-oriented language which uses generic function syntax is New Flavors.

- The support for implementation of and graceful coexistence with other object-oriented languages using the metaclass kernel.

The latter feature is particularly attractive, since it supplies a means whereby code in existing object-oriented languages can be reused, as well as provides an avenue for experimenting with new object-oriented languages.

The experiments presented in this paper used Portable CommonLoops (PCL), an implementation of CommonLoops designed to run on as wide a variety of Common Lisp implementations as possible. PCL has been available through the ARPAnet since January, 1986. The version of PCL released on 1/26/87 was used for the experiments reported in this paper.

The experiments are divided into two parts:

- An implementation of an X-based [Scheifler86] window system library. The implementation was not designed to test CommonLoops, but after it was finished, it seemed to represent a good candidate for assessing the usefulness of CommonLoops, since windowing and graphics applications are in some sense the canonical object-oriented application.
- An implementation of HP's CommonObjects [Snyder86a] using the CommonLoops metaclass kernel. This experiment was designed to test the utility of the metaclass kernel for implementing an objects language with substantially different inheritance semantics.

Extensive profiling measurements, undertaken to characterize the performance of CommonObjects on CommonLoops (COOL) *vis-á-vis* a native Lisp implementation of CommmonObjects, are also reported. From these profiling measurements, information was obtained which allowed COOL class definition time to be reduced by almost two orders of magnitude. This information was also used to pinpoint a bottleneck in the portable implementation that could profitably be eliminated by some machine dependent assembly code. The final section of the paper presents conclusions and suggests areas for further research.

2. Experience with the Language

The design and implementation of an object-oriented window library, called BeatriX, using CommonLoops permitted an assessment of the various language features to be made. The language features which will be discussed are the inheritance algorithm, the use of multimethods and generic function syntax, the partial integration with Common Lisp data types, the lack of method combination, and the lack of encapsulation.

2.1 The Inheritance Algorithm

As part of an effort to redesign the HP Common Lisp Development Environment [Cagan86], a new window system was desired. One of the goals was to base the window system on the new window system standard, X. Although X provides a common platform for all tools (including Common Lisp) to communicate with the display, the programmatic interface to X (Xlib) is fairly low level. An example of functionality that is difficult to achieve with Xlib is specializing a given type of window to achieve an application specific configuration.

The necessary flexibility can be achieved by using the "mixin" style of object-oriented programming [Weinreb81]. Mixins are sets of behaviors that can be combined using inheritance to achieve a specialized result. The mixin style of programming views classes as sets of nonhierarchical behaviors and inheritance as a means of combining them, with the intent of specializing some more general behavior [Cannon82]. This approach contrasts with the abstract data type style (supported by, among others, Smalltalk [Goldberg83], C++, and CommonObjects) in which classes are data types and inheritance specifies the structuring of the type lattice for subtyping[4].

An initial set of mixin classes for BeatriX was designed and is shown in Fig. 1. As the names indicate, the classes were divided into two groups: a set of basic window classes and a set of mixin classes for specializing the basic classes to a particular application. Mixins differ from regular classes in that, since they are intended to provide incremental functionality, they themselves must not be instantiated independently (similar to the abstract superclass idea of Smalltalk). In particular, instantiation should only proceed when the mixin is associated with a more general parent class. In old Flavors, mixins are supported by the options **:required-flavors** and **:mixture** to the flavor definition macro **defflavor** [Symbolics84].

As an example, consider trying to construct a class of windows having a border and title from the classes given in Fig. 1:

4. CommonObjects also provides a means whereby the programmer can modify the default subtyping algorithm.

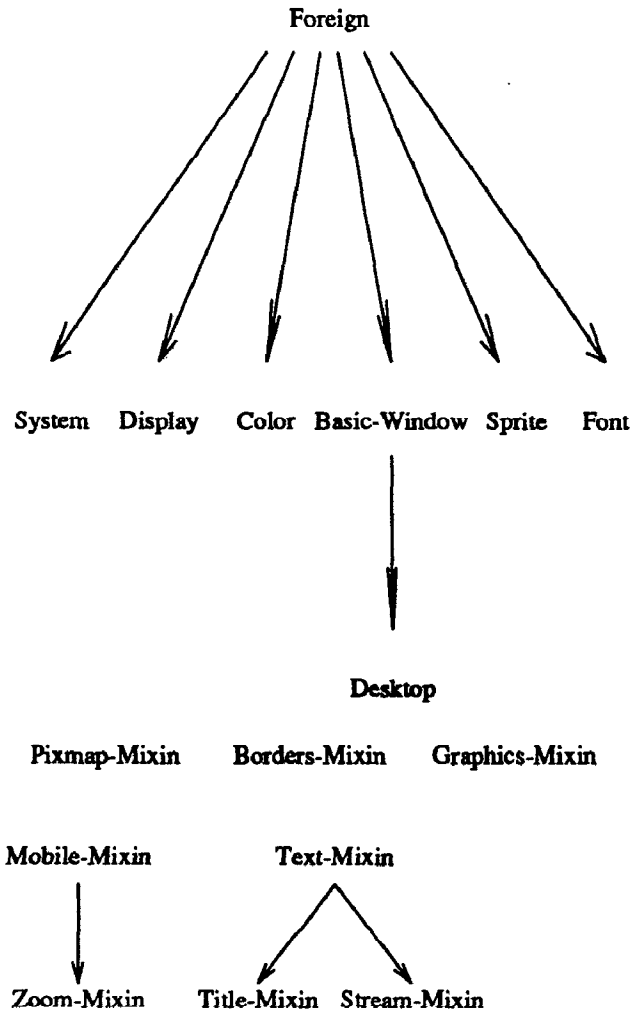

Figure 1. Window System Classes

```
(defclass
 (bordered-titled-window
  (:class class)
  (:include
   (borders-mixin title-mixin basic-window)
  )
 )

    < slot definitions >
)
```

A **bordered-titled-window** can also be build by constructing the inheritance hierarchy illustrated in Fig. 2, with the additional benefit of giving users the option of being able to instantiate a **titled-window** or a **bordered-window** if desired.

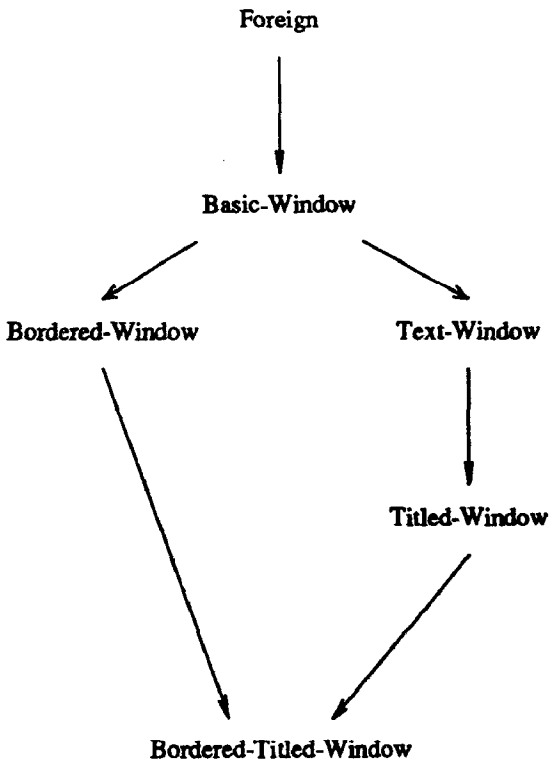

Figure 2. Inheritance for **bordered-titled-window**

CommonLoops computes a linearization of the inheritance tree, called the *class precedence list*, which is used during any operation involving a class' superclasses. As its name implies, the class precedence list determines which class in the inheritance chain has precedence when a conflict arises, for example, in determining which of a number of inherited methods to invoke. The CommonLoops algorithm for the default metaclass **class** produces the following class precedence list for the class **bordered-titled-window** illustrated in Fig. 2:

```
( bordered-titled-window  bordered-window
  titled-window  text-window
  basic-window  foreign
  T
)
```

The algorithm proceeds by doing a depth first traversal of the hierarchy, removing all but the last occurance of duplicate classes.

The classes are arranged with the most general superclass at the end of the class precedence list, and more specific classes closer to the subclass, so that methods defined on

the subclass will specialize the more general behavior of the superclass. Although the CommonLoops inheritance algorithm required structuring the code hierarchically, the effect of a nonhierarchical mixin system can be achieved.

Note that the CommonLoops inheritance algorithm is the same as for New Flavors but different from old Flavors. In old Flavors, the equivalent class precedence list would have been:

```
(bordered-titled-window  bordered-window
 basic-window  foreign
 vanilla  titled-window
 text-window
)
```

since the duplicates **basic-window** and **foreign** are eliminated except for the first occurance rather than the last. This class precedence list makes specialization difficult, since any methods defined by **titled-window** and **text-window** which are designed to specialize the more general behavior of **basic-window** will not do so.

2.2 Multimethods and Generic Function Syntax

One of the important innovations introduced by CommonLoops is multimethods. Multimethods are methods that dispatch on more than one argument. In addition to multimethods, CommonLoops uses generic function syntax to invoke a method, so a method invocation looks like a function call. The notion of a distinguished "self" parameter, as in Smalltalk or Flavors, disappears since dispatching can occur on any or all of the method's required parameters.

In general, the use of generic function syntax was viewed as a positive step, since it removed the syntactic distinction between object-oriented and functionally-oriented code. This was, in fact, one of the original goals of CommonLoops. In languages that originally were not object-oriented, like Lisp, use of a messaging operator requires the programmer to make an explicit decision when to use object-oriented constructs and when to use functional constructs. By moving to generic function syntax, the additional cognitive load of having to make this decision is eliminated. The programmer simply uses functional syntax for operation invocation, and can implement the operation as either a function or a method depending on what seems appropriate at a particular time during the development process. In addition, the standard Lisp debugging utilities can be used on method invocations.

The usefulness of multimethods was judged to be less clear. Most methods that discriminated on more than one argument in the window system application were so written primarily as an aid to type checking. Presuming CommonLoops classes and generic functions were integrated with Common Lisp's declaration mechanisms, multimethods would be redundant for this purpose, since the Common Lisp **declare** special form allows the argument and result types of functions to be declared.

Multimethods and generic functions in some sense remove the concept of a method being defined "on" a class, since the name space of method operations is no longer segmented by the class hierarchy, but rather through the Common Lisp package system[5]. This change reduces the usefulness of classes for implementing the medium scale (or module level) structure of a system, because the connection between methods and classes is broken. However, since the package system was designed to serve as the basis of modularization in Common Lisp, the designer can fall back upon it to group classes and operations under a particular package name, which then serves as the module.

One way to objectively assess the usefulness of generic functions and multimethods is to measure CommonLoops code and determine how often the system developer used these constructs. Measurements were made both on BeatriX and on the CommonLoops system itself. Since a large part of CommonLoops is written in itself, the latter would give a measure of how useful the developers found generic functions and multimethods. The results of these measurements are shown in Fig. 3 and 4.

In Fig. 3, the usefulness of generic functions is measured as the percentage of overloading on generic function names. A generic function name is overloaded if more than one method has the same name. The figure plots overloading as the percent of total generic function names that had one or more methods on the name's symbol. For the CommonLoops kernel, the great majority (89%) of the generic function symbols had a single method associated with them. In contrast, most of BeatriX's generic function symbols (67%) had two associated methods, and there were more generic function names that were heavily overloaded (up to 16).

These measurements suggest that function overloading was more useful during application development than during the implementation of CommonLoops itself. A caveat is necessary here, since many of the generic functions in the CommonLoops system are part of the

5. The package system establishes a mapping from print names to Lisp symbols, and thus serves to partition the name space for symbols.

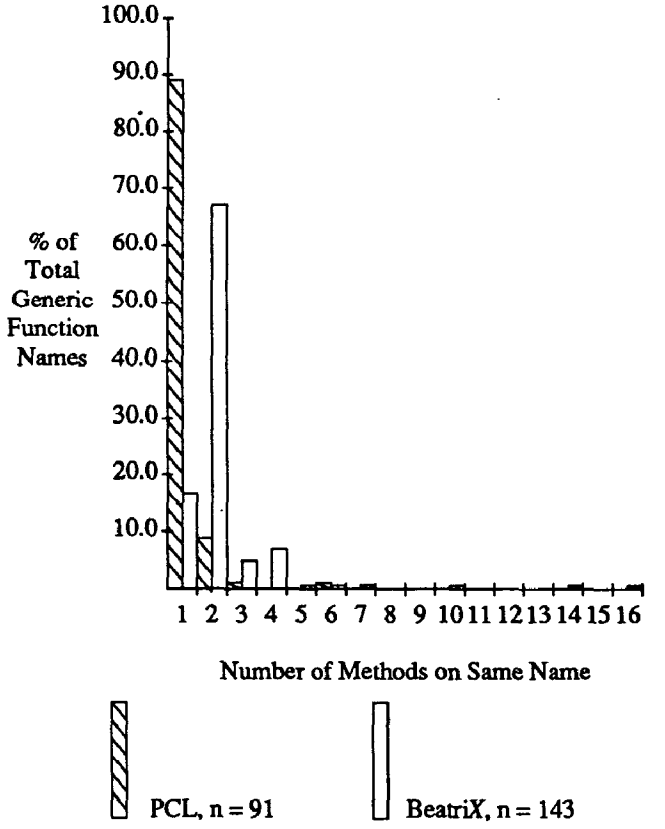

Figure 3. Function Overloading

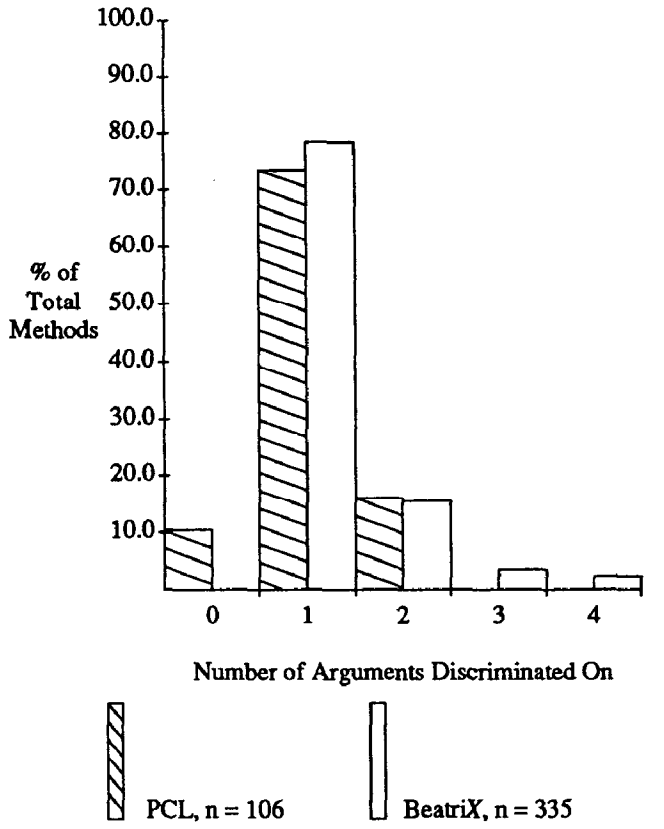

Figure 4. Multimethod Usage

metaclass kernel, and can presumably be specialized by a programmer implementing a new language through another metaclass. One measure of this hypothesis is to see how many new methods were defined on unoverloaded generic functions during the COOL development. Looking at the COOL implementation, only one method was defined on a CommonLoops metaclass method which previously was not overloaded.

Fig. 4 plots the results of measuring the percentage of total methods that discriminated on zero[6] or more arguments, and is thus a measure of how often the implementers used multimethods. As can be seen, in both the CommonLoops system itself and in the window system application, methods discriminating on a single argument, or classical methods, were used far more often than multimethods. Whether or not this was due to the implementers' experiences with classical methods in other object-oriented languages is open to question, but presumably as developers become more experienced with multimethods, multimethod usage may increase.

In the CommonLoops kernel, the maximum number of arguments discriminated on was two. Despite the limited use of multiargument dispatching, many of these methods are in a critical part of the method definition code where dispatching on both a discriminator object and a method object logically makes sense, since the operation to be performed may vary according to the classes of both the discriminator and the method. If multiargument dispatching were removed, either a dispatching class would be required or the methods on the method and discriminator classes would be required to differentiate using a case analysis on the class of the nonself argument. A more accessible example of how multimethods can simplify code when method behavior requires dispatching on two arguments is given in Section 5.

In contrast, as mentioned previously, most of the multiargument dispatching in BeatriX is used for type checking. An indication of this was the fact that only

6. A method discrimintaing on zero arguments is a default method, and is called if the argument classes do not match the specifiers for any other method with the same name.

11.1% of the methods defined to discriminate on more than one argument actually differed in the second or third argument from other methods. Many of the generic functions with more than one argument specifier had only one associated method; others were overloaded but only the first argument was relevent for dispatching. The other argument specifiers were the same for all methods with the same name. For those cases in which the second and third arguments were actually used to dispatch, the second and third argument were often used to discriminate between a method in which the arguments were not typed and a method in which type discriminators were specified.

The measurements of generic function and multimethod usage were made on the source code and therefore represent the static structure of the system. Dynamic monitoring of how often function overloading is really used during execution, and how often the second and third arguments are really needed during method discrimination would provide relevent data on whether the flexibility of late binding generic functions and multimethods is really needed during execution or whether early binding, type checking, and declarations could be used to remove the run time lookup. The static measurements reflect the usefulness of these concepts during the design and implementation of the system.

2.3 Partial Integration with Common Lisp Data Types

Another important feature of CommonLoops is the existence of classes that shadow certain of the basic Common Lisp data types. This feature allows method implementers to write methods that discriminate on objects of an underlying Common Lisp type, as well as on objects which are instances of a CommonLoops class. BeatriX used discrimination on Common Lisp types in 16% of its methods, exclusively on second or third arguments. As was mentioned above for multimethods, type checking was the major reason.

During the initial design of BeatriX, a heavier use of discrimination on Common Lisp types was anticipated. CommonLoops does not, however, allow CommonLoops classes to inherit from and thus specialize classes that shadow Common Lisp types. Thus the natural way of defining a stream that communicates with a window:

```
(defclass
 (window-stream
  (:class class)
  (:include (stream))
 )
)
```

is prohibited, because the built-in shadow classes for the Common Lisp types are of a different metaclass, and inheritance between metaclasses is forbidden. If such a class could be defined, then all the basic Common Lisp output functions could become generic functions, simplifying the coding of device independent presenters.

One possible solution to this problem would be to define a new metaclass that allowed inheritance from the built-in classes, perhaps as a mixin of the built-in metaclass and class. Exactly how to accomodate the desire for flexibility in specializing built-in classes is difficult to determine, because there are some built-in classes for which inheritance might be difficult to arrange or semantically meaningless. This is a result of the Common Lisp type system not being a true lattice. The type system in Common Lisp was deliberately designed without a partial order on all components to allow implementers freedom to implement some types in terms of others without having the implementation dependency show up in the type system.

2.4 Lack of Method Combination

Unlike Flavors, CommonLoops allows no daemon methods to be defined for a method. This eliminates complicated method combination procedures, but limits the ability of a programmer to add functionality to an existing method for which the source code was not provided. An example of where method combination would have been useful in BeatriX is for programming a counter to keep track of the number of window refreshes. In Flavors, a daemon method could be defined to maintain the counter:

```
(defmethod
 (basic-window :after :refresh) ()

 (incf *refresh-counter*)
)
```

In CommonLoops, the same effect requires specializing basic-window and adding a special refresh method which runs the super method:

```
(defmeth refresh ((w my-basic-window))

 (prog1
  (run-super)
  (incf *refresh-counter*)
 )
)
```

This solution will not work if the counter is to be incremented for all windows and not just for instances of **my-basic-window**[7]. To solve this, the user must modify the source code of **basic-window**.

2.5 Lack of Encapsulation

CommonLoops also provides no facilities for hiding parts of a class representation from a user. The function **get-slot** provides slot access even outside a method. Accessor functions for slots are also available globally, within the name space established by the package where the class is defined.

The lack of encapsulation presented a problem with software change management in BeatriX. In version 10 of X, there is a large degree of inconsistency between when to use a color map register and when to use a solid color pixmap as an actual parameter to an X function. In BeatriX, it was necessary to have both a slot for the background color register and the background color pixmap in the **basic-window** class. In version 11 of X, much of this inconsistency is eliminated, making it possible to remove one of these instance variables. However, the ready availability of accessor functions allows users to gain access to and thus become dependent on the implementation of BeatriX's internals. BeatriX implementers are therefore dependent upon convention rather than enforced language mechanisms to avoid having internals leak out into applications.

Another more fundamental problem with the lack of encapsulation involves inheriting from classes shadowing the built-in types. Subclasses in CommonLoops have access to the full internal definition of their supers, including slot accessors. It is not possible, for example, to encapsulate a superclass, so that a subclass can inherit from it, but will not inherit slot accessors. Superclass designers can make no assumptions about what data remains hidden. In particular, with implementation dependent classes such as **stream**, lack of encapsulation between a subclass and its superclass means that inheritance must be forbidden *a priori*.

3. Experience with the Metaclass Kernel

An important feature of CommonLoops is the metaclass kernel. Unlike Smalltalk and Objective-C, metaclasses in CommonLoops are neither automatically generated when a class is defined nor are they generators for instances of a class, but rather are defined by a language implementer to support a particular embedded object-oriented language. Different metaclasses can be used to implement different inheritance algorithms, different storage allocation strategies for instances, and different method dispatch strategies. Classes in the CommonLoops language are instances of the default metaclass **class**. Metaclasses offer both a means of maintaining compatibility with existing object-oriented languages and a supporting base for experimenting with new languages.

To assess the usefulness of the metaclass kernel for supporting different object-oriented languages, the CommonObjects language was implemented using the metaclass kernel. The following subsections discuss the important features of the implementation.

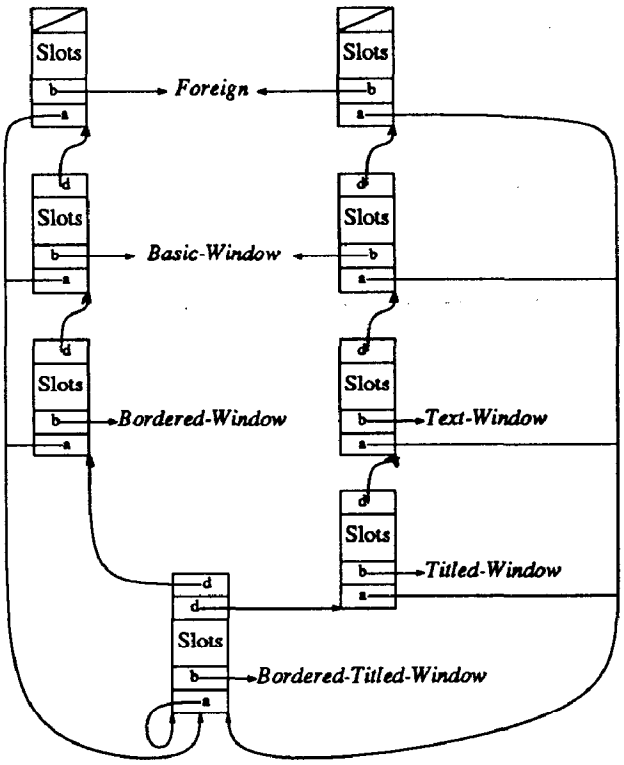

Figure 5. Structure of a Single CommonObjects **bordered-titled-window** Instance

3.1 Inheritance Semantics

CommonObjects supports a very different inheritance semantics from CommonLoops. In Fig. 5, the structure of a single COOL **bordered-titled-window** instance with the same inheritance as in Fig. 2 is illustrated. The italicized names correspond to class objects, and the letters are keyed to the list of instance parts at the end of

7. The CLOS standard will include method combination, as a contribution of New Flavors.

this subsection. Notice that a set of slots for the shared ancestor superclasses **basic-window** and **foreign** is duplicated along the two inheritance branches through which **bordered-titled-window** inherits **basic-window** and **foreign**. In a CommonLoops instance with this inheritance, the two occurances of **basic-window** would be merged, as was discussed in Section 2.1. CommonObjects inheritance is *tree-structured*, as opposed to the linearizing inheritance of Flavors and CommonLoops, since no attempt is made in CommonObjects to eliminate duplicate state during processing of a class definition. Another form of inheritance, *graph-structured* inheritance, merges state but maintains seperate methods along diverging inheritance branches, so duplicate method invocations are possible. Graph-structured inheritance is used by Smalltalk with multiple inheritance [Borning82] and Trellis/Owl [Schaffert86]. The selection of tree-structured inheritance in the CommonObjects design grew from a desire to maintain encapsulation between superclasses and their inheriting subclasses [Snyder86b].

In addition, CommonObjects supports *instance centered encapsulation*, a style of encapsulation in which methods can only access instance data in instances on which the method was invoked (similar to Smalltalk, Objective-C, and Flavors). A looser style of encapsulation, *class centered encapsulation*, allows methods to access data in any instance of the class on which they were defined. CLU [Liskov77] and C++ support this kind of encapsulation. CommonLoops, as mentioned above, has no encapsulation whatsoever, since any method or function can access instance data if the slot name or accessor function name is known.

CommonObjects enforces a stricter kind of encapsulation than other instance centered languages, since inheriting classes cannot access superclass slots without invoking a method. In other instance centered languages, like Smalltalk and Objective-C, superclass instance variables are lexically accessible within methods defined on the class. A form of syntatic sugar is provided to simulate lexical reference in CommonObjects, but the result is translated into a method invocation, and if the user redefines the superclass accessor method, the new accessor method will be invoked.

Given the large difference in inheritance semantics, CommonObjects would seem to provide a challanging test of the flexibility of the CommonLoops kernel. Suprisingly, the inheritance semantics were the easiest aspect of the CommonObjects specification to implement, mainly because the CommonLoops class precedence list calculation was bypassed and inheritance was maintained separately by COOL. An instance of an inheriting class consists of multiple parts, one for each direct superclass, recursively to the top of the superclass tree. For example, as shown in Fig. 5, the **bordered-titled-window** instance has the following parts:

a. A pointer to the object piece corresponding to **self**,

b. A pointer to the class object,

c. A part containing storage for the instance's slots,

d. A pointer to an object piece for each direct superclass.

3.2 CommonObjects Methods

Since CommonObjects classes do not use the CommonLoops class precedence list for method inheritance, method inheritance must be arranged differently. In fact, CommonObjects semantics require method inheritance to be computed at compile time, so method inheritance in COOL is done by automatically defining a method on the inheriting class which simply calls the superclass method directly.

The CommonLoops class precedence list is used, however, for the "universal methods." CommonObjects semantics require that all CommonObjects classes have a set of standard methods defined on them at class definition time, which implement certain universally useful operations [Snyder86a]. The class precedence list for all CommonObjects classes is ordered with the class itself first, then the CommonObjects metaclass (**common-objects-class**), then the CommonLoops class **object**, which is a specifier for some of the kernel methods, such as **print-instance**. The CommonLoops class precedence list for a CommonObjects class having the inheritance structure in Fig. 2 would be:

```
( bordered-titled-window
  common-objects-class
  object
  T
)
```

Thus the CommonObjects inheritance tree is maintained seperately from the CommonLoops class precedence list. The set of universal methods can, in effect, be shared by all CommonObjects classes by defining them on the metaclass **common-objects-class**. Only the **:initialize-variables** method need be generated on a per class basis, since programmers are allowed to specify custom initialization code for slots and CommonObjects semantics require that the code execute in the context of a method definition. This arrangement modifies the semantics of undefining a CommonObjects method

somewhat, since a user can no longer undefine a default universal method. Redefining a universal method on a class, or undefining a universal method that has been redefined is still valid, however.

3.3 Difficulties

The lack of hooks into the PCL code walker made the implementation of a number of features impossible. One of them was lexical reference to parent slots. Checking for improper use of the distinguished lexical variable **self** within a method, and the checking of slot types were also not implemented because of the code walker difficulties.

Another area where CommonObjects semantics has been lost is in the ability to define methods that have varying numbers of required parameters. Since the number of required formal parameters must match between CommonLoops methods with the same generic function name, the ability to define methods with differing numbers of method parameters is no longer possible[8]. CommonObjects methods are simply implemented as CommonLoops generic functions, so the CommonLoops restriction must be propagated to them as well.

The generic function style of programming is somewhat different from the message operator style, which CommonObjects supports. In the generic function style, name clashes are avoided by convention and use of the package system, while the message operator style avoids clashes by partitioning the method name space on the basis of class name. Clashes are possible using the message operator style if one tries to define a subclass with two superclasses having inconsistent definitions of a method. Implementing the message operator style on top of the generic function style increases the potential for clashes. For example the **move** operation applied to a file object might take only one argument, the name of the new file, while the same operation applied to an icon might require two, the x and y co-ordinates of the new location. With the generic function style, either one of the two operations would have to be renamed or put in a separate package, or the operation to move an icon could be redefined to take a single point argument, which is then decomposed into co-ordinates within the method.

A more serious restriction resulted from underspecification of the compile time semantics of CommonLoops. CommonObjects uses a form of case analysis called "moderation compilation" [Creech85] to carefully control when a class or method definition is replaced in the compile time environment. Enough information about a class or method is maintained in the compilation environment so that method inheritance between a superclass method and an inheriting subclass defined in the same file will compile correctly. CommonLoops methods are not defined until load time. As a result, it is not possible to compile the COOL methods for a superclass and the class definition for an inheriting subclass in the same file, since the superclass methods are not available until load time. In addition, since CommonLoops fully defines classes as objects at compile time, any preexisting definition will be destroyed.

Perhaps the most serious problem with implementing COOL was the inability to implement the CommonObjects **call-method** or **apply-method** correctly. These forms provide a means whereby a method can call another method on a class or one of the direct superclasses without going through method lookup. Implementing the semantics requires that it be possible to create compiled code which will reference a symbol that can only be created at load time. Although Common Lisp has a reader macro that is supposed to arrange for execution at load time, this functionality is undefined if the code is being processed by the compiler. Common Lisp also has a top level form, **eval-when**, that allows load time control over evaluation of other forms at the top level. The lack of a Common Lisp function to execute a form at load time within macro generated code required the implementation of a very fragile solution depending on equivalence of interned symbols in the compile and execution environments. The solution has the serious drawback that name clashes could occur in the symbol names, although the names are chosen to avoid clashes as often as possible.

3.4 Size

The total number of noncommented source lines in COOL is 1842, 6133 including the PCL kernel. This compares favorably with the native Lisp implementation of CommonObjects, which is 4726 noncommented source lines. For a slightly larger system, COOL provides most of CommonObjects semantics with the additional functionality of the CommonLoops language, and the extensibility of the CommonLoops kernel. Thus COOL gives programmers more options, since those who want to program in either the encapsulation style of CommonObjects or the mixin style of Flavors/CommonLoops can easily do so.

3.5 Portability and Availability

In addition to running on HP Common Lisp, in which it was developed, COOL has been ported to Kyoto Common Lisp[9]. COOL was designed to be as portable as

8. The CLOS could potentially allow this difference to be finessed through method combination.

PCL, and therefore should run on any Common Lisp that runs PCL. A series of regression tests and performance tests are included with the distribution. The PCL version on which COOL currently runs is also included in the distribution, to avoid software tracking problems.

4. Performance

Performance analysis helped to identify problem areas in COOL and also served as a useful benchmarking mechanism to obtain a rough comparison of COOL performance with the Portable CommonLoops language implementation and with the native Lisp CommonObjects implementation. As an example of how performance analysis helped improve COOL, an initial naive implementation of COOL generating universal methods on a class by class basis was almost three orders of magnitude slower during compilation than the native CommonObjects. A solution involving the CommonLoops class precedence list, as described in Section 3.2 above, allowed default universal methods to be defined on all CommonObjects classes via **common-objects-class.**

The hardware configuration used during performance testing was an HP9000/Series 320 68020 workstation, running at 16.5 MHz with 6 megabytes of main memory. The tests were run with the HP-UX operating system in state 1, so no other background processes were in operation, and the Common Lisp image size was 10 megabytes. The paging disc was an HP 7914. A special 10 μsec clock was used to obtain the measurements, so the measured times should be accurate to 10 μsec. Both the CommonLoops language and COOL were compiled with the maximum portable optimization on (e.g. (optimize (speed 3) (safety 0))), the native CommonObjects was as distributed with the HP Common Lisp Development Environment [HP86].

The tests were executed by using a Common Lisp macro to generate a function that performed the operation to be tested 20 times, and the function was subsequently compiled and executed. No compiler optimizations were turned on for the tests, except those that the CommonLoops language and COOL locally enabled. Both the compilation time and execution time of the function were measured. Care was taken to avoid garbage collection by garbage collecting before the measurements, to avoid paging by doing the full test (i.e. 20 iterations) before a series of the same tests, and to prime the workstation's cache by doing each operation once before the measurements were made.

All measurements are given as the ratio of the time taken by an operation for COOL or the CommonLoops language to the time taken by the same operation for the native Lisp code implementation of CommonObjects.

4.1 Definition Performance

The performance of class and method definition was measured for all three implementations. Class definition operations consisted of defining a class with zero, one, two or three slots but no parents, then with one, two, or three parents but no slots. For each implementation, the default slot accessibility was used, thus slot accessors were not generated for COOL or CommonObjects but were for the CommonLoops language. Method definition operations consisted of defining a method for which zero, one, two or three methods of the same name existed for different classes. The generated CommonLoops language method was classical (i.e. not a multimethod).

Operation	Slots	Supers	COOL/CO	PCL/CO
Define Class	0	0	0.30	0.93
	1	0	0.47	1.88
	2	0	0.67	2.61
	3	0	0.76	3.44
	0	1	0.89	1.04
	0	2	0.95	0.94
	0	3	1.04	0.87
Create Instance	0	0	57.00	24.52
	1	0	7.14	4.55
	2	0	7.55	4.69
	3	0	3.39	2.25
	0	1	12.96	4.90
	0	2	8.93	2.39
	0	3	21.22	4.15
Inherited Operation Invocation	-	0	5.67	4.83
	-	1	11.14	4.29
	-	2	13.57	4.86
	-	3	16.00	4.43

TABLE 1. Class Definition, Instance Creation, and Inherited Operation Invocation

The results for class definition are tabulated in the upper part of Table 1 while the upper part of Table 2 shows the results for method definition. COOL compares favorably with CommonObjects, as does the CommonLoops language, with the exception of the increase in class definition time as the number of slots increases for the CommonLoops language, due to the generation of

9. COOL is currently available through anonomyous FTP from **ingres.berkeley.arpa** in the directory /pub/cool or by electronic mail request from **cool@hplabs.**

accessor functions. Presumably, a similar increase would occur in COOL and CommonObjects if slot accessors were requested during class definition.

An explanation for the greater amount of time involved in class definition for CommonObjects as opposed to the CommonLoops language and COOL can probably be found in the richer compilation semantics of CommonObjects, as explained in Section 3.3. Moderation compilation takes longer than simply defining the class outright. As mentioned above, CommonLoops compilation semantics simply defines a class fully at compile time, and does not fully define a method until load time. The extra amount of checking required to implement moderation compilation makes CommonObjects definition operations somewhat slower than COOL or the CommonLoops language.

It should be noted that redefinition times for COOL classes were more than 30 times slower than the initial definition times. This was not true of either the CommonLoops language or CommonObjects, and is probably due to the algorithm in the COOL implementation used to determine whether an incompatible change to the class occured. Some room for improvement exists there.

Operation	Functions	COOL/CO	PCL/CO
Define Operation	0	0.57	0.45
	1	0.70	0.46
	2	0.76	0.50
	3	0.83	0.49
Operation Invocation	1	5.67	4.83
	2	25.67	24.67
	3	19.38	18.75
	4	22.86	21.43

TABLE 2. Method Definition and Operation Invocation

4.2 Method Invocation and Instance Creation Performance

The method invocation test consisted of invoking a method when one, two, three, or four methods were defined on the same generic function symbol. Since PCL caches most recently used methods in the generic function, a special strategy was used to avoid simply measuring the speed of a cached invocation and thus to obtain measurements of method lookup time. When more than one method was defined on the generic function, invocation proceeded sequentially through instances of the various classes. This assured that the PCL method cache would be cleared and the actual method lookup time would be measured, rather than simply measuring the time to invoke a cached method. The inherited method invocation test consisted of invoking a method inherited through zero, one, two or three parents. Times for compiling the invocation form are not included in the measurements, since they were the same for all three systems. The instance creation test was performed by simply creating 20 instances of a class with the appropriate number of slots and parents.

The results of the method invocation tests are tabulated in the lower parts of Tables 1 and 2. Clearly, for method invocation, the native Lisp implementation of CommonObjects is the better performer, especially for inherited methods, by factors of from 5 to 25. The superior performance of CommonObjects can probably be ascribed to a number of factors. CommonObjects uses some special assembly code to implement inherited methods, while COOL simply generates a Common Lisp function that calls the superclass method. An assembly coded function is also used during method dispatch in CommonObjects. In addition, the portable dispatch function in PCL is not properly tail recursive, so optimization of the actual method function call to a direct jump, without pushing the return address on the stack, is not done by the compiler.

For instance creation, the native Lisp CommonObjects also performs better. However, the performance of COOL and CommonLoops improves as the number of slots increases. As the number of parents increase, the timings of both COOL and CommonLoops become slower, though COOL becomes slower faster than CommonLoops. A possible reason is that the initialization protocol in COOL requires more messaging than in CommonLoops, so more time may be transpiring during messaging.

In an effort to improve method dispatch performance, a dispatch function for classical methods was hand coded in assembler and the implementation of the method table in the generic function discriminator was changed from being an association list to a fixed size hash table. Additionally, the underlying representation of classes (but not instances) was changed so every class has a unique, 28 bit id which is used as a hash key. The resulting speedup made method dispatch between 20% to 50% faster than CommonObjects for classical methods. Since most of the PCL kernel is implemented using methods, the speedup of method dispatch had a significant effect on the performance of the entire system.

5. Conclusions and Suggestions for Further Research

Although most of the code for handling inheritance in COOL was written with minimal use of the

CommonLoops kernel, the metaclass protocol provided a supporting mechanism for maintaining information about defined methods and classes. This information could be useful in a programming environment, for example, since the kernel provides a standardized interface for accessing information about all object-oriented languages in the environment. For implementing a language with linearizing inheritance, more of the metaclass kernel could probably be used. Perhaps most importantly, the ability to provide two languages with different inheritance semantics in the same system allows users to choose a particular style of inheritance to fit their application, while still allowing the two languages to share some common code. The performance figures indicate that generic functions should be efficiently implementable on conventional architectures such as the MC68020, though the efficiency of multimethods remains an open question.

The results of the code measurments reported in Section 2.2 suggest that generic functions were more heavily used in BeatriX than in the CommonLoops kernel itself. For multimethods, the code measurements imply that discrimination on a single argument, or classical methods, was more heavily used than discrimination on multiple arguments. An examination of the CommonLoops method definition code suggests, however, that multimethods might actually be more important than the statistics imply, and that the statistics may rather be a reflection of prior programmer experience. In addition, multimethods in the CommonLoops kernel may contribute to making the system more extendable.

Another example of how multimethods might be even more useful than first appears comes from the realm of user interfaces. One of the often mentioned advantages of object-oriented programming for user interfaces is that a display method can be defined on a class, and thus objects of various classes can display themselves in a class dependent manner without requiring the programmer to use extensive case analyses to determine how to display something. The Smalltalk Model-View-Controller [Goldberg84] evolved as a response to this need, but segregating functionality into these three categories has proven difficult. The difficulty is that displaying an object requires knowing the particular characteristics of the output device, as well as the characteristics of the object to be displayed. Displaying an object in a text window requires different low level actions from displaying it in a graphics window, for example. Users of traditional object-oriented languages are therefore required to write a case analysis into the model or display classes to handle different types of displays. An interesting direction for future research would be to explore the use of multimethods in programming user interface systems.

The analysis in Section 2 brings up a point related to computer language design and engineering. While most language designs are subject to a prototyping phase, in which a small community of trial users implement in the language and give feedback to the language designers, language designers rarely attempt to quantitatively access which constructs were most useful and which were less so, or if particular idioms could be built into the language to save programmer effort. Most trial user feedback comes in the form of qualitative comments. Often it is difficult to judge whether these comments are peculiar to the particular group of trial users, or whether the comments have wider applicability.

As languages become more and more complex and evolve further and further away from being models of the underlying processor, it becomes more important to quantitatively access the usefulness of particular language features. Removal of rarely used features in the early stages of a design can save later implementers much extra effort. Similarly, identification of features that appear on the surface to be relatively unused but which, when used, perform a very crucial function (as is the case with multimethods) can suggest areas where extra documentation and programmer education may be needed. In addition, identification of commonly occuring language idioms may suggest areas where the language design can be augmented to provide additional functionality.

Developers of new languages might therefore consider building statistical collection code into their translation software tools, and, after properly informing their trial user community, have the results summarized and mailed to them periodically to facilitate collection of quantitative information on language construct usage. Such a procedure, commonly followed in other engineering disciplines, could allow language designs to evolve on a more quantitative basis.

Acknowledgements

We would like to thank Larry Rowe and his Objfads study group, for letting us use **ingres** to distribute the COOL software and the ANSI Common Lisp Object System standardization subcommittee, for listening to our comments on the metaclass kernel. Special thanks go to Gregor Kiczales, whose valiant efforts against the Dragon of Importability have yielded the Sugar Magnolia: a flexible, portable object-oriented programming system.

References

[ANSI87] Daniel Bobrow, David Moon, et al., "Common Lisp Object System Specification," ANSI X3J13 Document 87-002, American National Standards Institute, Washington, DC, 1987.

[Ballard83] Stoney Ballard and Stephen Shirron, "The Design and Implementation of VAX/Smalltalk-80", in **Smalltalk-80: Bits of History, Words of Advice**, Glenn Krasner, ed., pp. 127-149, 1983.

[Bobrow86] Daniel Bobrow, et al., "CommonLoops: Merging Common Lisp and Object-Oriented Programming," Proceedings of OOPSLA, **SIGPLAN Notices, 21(11)**, pp. 17-29, 1986.

[Borning82] Alan Borning and Daniel Ingalls, "Multiple Inheritance in Smalltalk," **Proceedings of AAAI**, pp. 234-237, 1982.

[Cagan86] Martin Cagan, "An Introduction to Hewlett-Packard's AI Workstation Technology," **Hewlett-Packard Journal**, Vol. 37(3), pp. 4-14, 1986.

[Cannon82] H. I. Cannon, "Flavors: A Non-Hierarchical Approach to Object-Oriented-Programming," 1982.

[Cox86] Brad Cox, **Object-Oriented Programming**, Addison-Wesley, Reading, MA, 274 pp., 1986.

[Creech85] Michael Creech, "The Compile Time Environment," STL Internal Memorandum, 1985.

[Falcone83] Joseph R. Falcone and James Stinger, "The Smalltalk-80 Implelmentation at Hewlett-Packard," in **Smalltalk-80: Bits of History, Words of Advice**, Glenn Krasner, ed., pp. 79-112, 1983.

[Goldberg83] Adele Goldberg and David Robson, **Smalltalk-80:The Language and Its Implementation**, Addison-Wesley, Reading, MA, 714 pp., 1983.

[Goldberg84] Adele Goldberg, **Smalltalk-80:The Interactive Programming Environment**, Addison-Wesley, Reading, MA, 1984.

[HP86] **Lisp Programmer's Guide**, Hewlett-Packard Co., 1986.

[Liskov77] Barbara Liskov, et al., "Abstraction Mechanisms in CLU," **Communications of the ACM**, Vol. 20(8), pp. 564-576, 1977.

[McCullough83] Paul McCullough, "Implementing the Smalltalk-80 System: The Tektronix Experience," in **Smalltalk-80: Bits of History, Words of Advice**, Glenn Krasner, ed., pp. 59-78, 1983.

[Moon86] David Moon, "Object-Oriented Programming with Flavors," Proceedings of OOPSLA, **SIGPLAN Notices, 21(11)**, pp. 1-8, 1986.

[Schaffert86] Craig Schaffert, et al., "An Introduction to Trellis/Owl," Proceedings of OOPSLA, **SIGPLAN Notices, 21(11)**, pp. 9-16, 1986.

[Scheifler86] Robert Scheifler and Jim Gettys, "The X Window System," MIT LCS Memo LCS-TM-368, Massachutsetts Institute of Technology, Cambridge, MA., 1986.

[Snyder86a] Alan Snyder, "CommonObjects: An Overview," **SIGPLAN Notices, 21(10)**, pp. 19-28, 1986.

[Snyder86b] Alan Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages," Proceedings of OOPSLA, **SIGPLAN Notices, 21(11)**, pp. 38-45, 1986.

[Steele84] Guy Steele, **Common Lisp: The Language**, Digital Equipment Corp., 465 pp., 1984.

[Stroustrup86] Bjarne Stroustrup, **The C++ Programming Language**, Addison-Wesley, Reading, MA, 327 pp., 1986.

[Symbolics84] "FLAV Objects, Message Passing, and Flavors," Symbolics, Inc., 1984.

[Weinreb81] Daniel Weinreb and David Moon, **Lisp Machine Manual**, Symbolics, Inc., 1981.