PCLOS: Stress Testing CLOS

Experiencing the Metaobject Protocol

Andreas Paepcke
Hewlett-Packard Laboratories
1501 Page Mill Rd.
Palo Alto, CA 94304
paepcke@hplabs.hp.com

Abstract

This paper demonstrates that the CLOS *metaobject protocol* approach to defining and implementing an object model is very powerful. CLOS is an object-oriented language that is based on Common Lisp and is in the process of being standardized. Implementations of CLOS are themselves object-oriented with all major building blocks of the language being instances of system classes. A metaobject protocol provides a framework for CLOS implementations by specifying the hierarchy of these classes and the order and contents of the communication among their instances. This design has made CLOS both flexible and portable, two design goals that traditionally conflict. In support of this suggestion we present a detailed account of how we added object persistence to CLOS without modifying any of the language's implementation code.

1 Introduction

We will begin this paper with some introductory thoughts about the problems of building systems that are both flexible and portable. This will be followed by the suggestion that CLOS, a new object-oriented programming language, exemplifies a solution to the apparently irreconcilable conflict between the design goals of flexibility and portability.

This claim is then supported with a detailed, nuts-and-bolts account of some of the author's efforts to build PCLOS [1, 2]. The PCLOS system adds database-independent persistence to CLOS objects and introduces recovery into the language. That account will be limited to those aspects of the design that made extensive use of the CLOS Metaobject protocol (MOP) mechanism. This mechanism defines the internal static structure of the language, as well as the dynamic, run-time behavior. It makes it possible to mold the language's object model and implementation to a very high degree.

This detailed account requires some prior introduction to CLOS, the metaobject protocol and PCLOS. Section 2 therefore introduces enough material about the CLOS language itself that the following section on the metaobject protocol can be understood. Section 4 serves a similar purpose, in that it introduces enough information on PCLOS that the following section on the manipulation of the CLOS metalevel for its implementation can be understood. Reference [1] describes PCLOS in more detail. Reference [2] discusses the PCLOS design and uses it to explain some issues involved in object persistence.

It is generally agreed that the software component of our systems needs to change over time. The frequency and extent of change differs for the various system components. In particular, it tends to be different for databases and their management systems on one hand, and application programs and their environment on the other.

All of our efforts in maintaining flexibility to accommodate this change are affected by what have been fundamental tensions in the design of systems. Figure 1 illustrates this problem.

Ideally we would have our systems be both portable and flexible at the same time. Unfortunately, these goals are generally in conflict with each other. Designing for portability leads us to *constraining* systems to ensure the ability of mapping onto the weakest targets. The final stage of this tendency is usually *standardization.* This is contrasted by the tendencies induced by the flexibility goal. This goal will tend to push the design towards *expansion*, towards open, highly customizable systems. Instead of standardization, the goal of flexibility has a *randomizing* effect.

But a reconciliation of the conflicting design goals of flexibility and portability can be achieved. The solution is based on the realization that if the mechanisms for changing a portable system are themselves part of that system, then the ideosyncrasies of disparate design instantiations will port along with the system.

CLOS is a full-scale attempt to reconcile portability and flexibility. Its implementation is the focus of the more specific, detailed remainder of this paper.

2 Some CLOS Facts

This paper cannot provide an in-depth tutorial of CLOS. Reference [3] is the 'official' reference for the CLOS programmer. It explains the concepts and contains manual pages for all the built-in operations. Reference [4] is an excellent introduction with a good balance between formal and informal presentation. It also contains many useful examples.

CLOS consists of the standard Common Lisp [5] with an object model added to it. We now outline the central concepts in just enough detail to prepare for a subsequent look at the insides of the language.

2.1 Generic Functions and Methods

Like other object-oriented languages, CLOS features late binding of procedure names to pieces of executable code. This dynamic dispatch is done through so-called *generic functions* which are CLOS' way of "sending messages to objects", a concept that is in one form or other common to all object-oriented programming languages.

Generic functions, when called, will examine the classes of the arguments passed in the call. Based on these classes they will transfer control to an appropriate body of code which is called a CLOS *method.*

Figure 1: Tensions in System Design

2.2 Method Combination

In addition to class-sensitive method selection, CLOS allows the programmer to specify "roles" for methods. These roles will also impact the chain of events that comprises the execution of a generic function. Some roles are predefined in CLOS, new ones may be specified by CLOS programmers.

Two of the built-in roles are called `:before` and `:after` methods. They are supplied with CLOS because they cover the frequently needed case of having some pieces of code run before or after some *primary* method. Examples are preparation and cleanup for file access activities, initiating output of diagnostic messages during debugging sessions, or primary method access restrictions.

Selecting and running the methods with different roles in the correct order is called *method combination.*

3 The CLOS Metaobject Protocol

The CLOS language consists of five major concepts:

- Classes
- Slots
- Generic functions
- Methods
- Method combination

Classes contain the descriptions of instance structure and behavior. **Slots**[1] are the 'containers' of data in an instance and hold administrative and other information

[1] *Slots* are often called *instance variables* in other object systems.

about that data. **Generic Functions** are the mechanism through which run-time, class-sensitive dispatching is accomplished. **Methods** are pieces of code which are associated with generic functions and contain the code the generic functions will dispatch to. **Method combination**, finally, enables CLOS programmers to structure and augment the execution of methods in useful ways.

The CLOS metaobject protocol describes the design of the operations and interactions of these five building blocks in an object-oriented fashion. Its purpose is to clearly define the semantics of the language in a standard that allows significant, yet portable changes to it. The protocol is documented in [6, 7] and we now present a simplified summary of its important pieces.

Since the building blocks are implemented as an object-oriented program, their implementation is specified by a collection of classes referred to as *metaobject classes*. The metaobject class that describes CLOS classes is sometimes called a *metaclass*. Instances of metaobject classes are called *metaobjects* and the details of the communication among these are called *protocols*. The CLOS metaobject protocol therefore consists of:

1. Public hierarchies of metaobject classes whose instances implement the five building blocks of the CLOS language.
2. A public protocol which describes when metaobjects are created, which messages must pass between them, and which parameters are to be used.

Given this information, a systems programmer may radically extend or modify CLOS by subclassing metaobject classes and by selectively shadowing methods that operate on their instances.

Note that subclassing and shadowing at the metalevel does not require any modifications of existing system code or even access to system sources! These mechanisms are thus non-intrusive and will work on any proper CLOS implementation. This, together with the fact that the meta-level of CLOS is intended to be standardized, is the reason why far-reaching modifications or extensions to CLOS are portable[2].

Figure 2 illustrates the relationship between the metaobject protocol, the CLOS language and a programmer. For the programmer, the concepts of the language are fundamental entities whose behavior is laid down in the CLOS manual. In reality, these entities are instances of classes at the metalevel.

In addition to portability, this has many advantages for systems programmers: the object-oriented design carries with it all of the positive effects associated with the paradigm. In addition, the ability of objects to provide information about themselves through well-defined interfaces makes it trivial for an expert to cleanly learn 'systemy' details about an application's classes and their methods or slots.

In order to work with the MOP to modify CLOS, a systems programmer must, of course, understand the object-oriented language that is used at this metalevel to define metaobject classes, to instantiate them and to send messages among their instances. It is a very elegant twist that this language is itself – CLOS. After some bootstraping, the language in which CLOS implementations are written is therefore CLOS. This approach of using the product of a system's implementation to construct that very product is called *metacircular*. The metacircularity of CLOS is very useful because it implies that a systems programmer does not need to learn a new language, and that CLOS proficiency may be leveraged for metalevel comprehension and programming.

We now present an example of using the metaobject protocol for a non-trivial modification and extension of

[2] The standardization is not yet complete. This paper is therefore based on a snapshot of the standard.

Figure 2: Metalevel Programming Produces the CLOS Language

CLOS. This example is PCLOS, a system that makes CLOS persistent on a variety of data stores, and which introduces recovery and queries into the language.

4 A Quick PCLOS Overview

This section very briefly introduces PCLOS. We include only aspects that are necessary to understand the material in section 5 on the details of integrating persistence into CLOS.

PCLOS is an experimental system that allows CLOS objects to persist beyond the Lisp session that created them. This is accomplished by mapping CLOS objects into information that is stored in a database or object server. It is possible for objects to be stored in one of several different databases of which PCLOS currently supports three. Figure 3 shows a top-level view of the PCLOS architecture. This architecture is general enough to accommodate several different languages, but we have so far worked exclusively with CLOS.

The layer between the databases and the programming language provides the insulation that is needed for making the system database-independent. It is called the *virtual database* layer because it provides a data model onto which the objects of the programming language are mapped. Separate mappings are then provided from the virtual database onto each of the databases to be supported. In particular, schemas are automatically generated to reflect the language-level class structure, and slot access and queries are translated into operations native to each respective database.

Users express queries in terms of their programming lan-

Figure 3: Top-level View of the PCLOS Architecture

guage. PCLOS then compiles these queries into expressions appropriate for underlying databases. Figure 4 shows an example.

The *protocol adapters* in figure 3 make it possible to take advantage of advanced features of individual databases which go beyond what is needed to implement the virtual database. Examples of this are richer queries which might allow multi-valued results, or an ability to define new operations for an underlying database [8].

PCLOS attempts to provide object persistence transparently whenever possible. This policy has been suspended whenever database-related capabilities other than persistence were introduced into the programming language, or when efficiency considerations require programmer knowledge. Examples of the former are transactions and queries, an example of the latter is the declaration of individual slots to be transient.

5 PCLOS and the MOP

In this section we show how the metaobject protocol has been used to build PCLOS without any modification to the system code. Language-internal manipulations are necessary to satisfy the following requirements:

- Access to instance slots must be intercepted so that values can be retrieved from or written to the databases.
- Some persistence-related information must be kept with each instance.
- Class hierarchies and the details of individual classes must be inspectable so that correct schemas can be generated for the various databases and queries over CLOS objects can be translated to database-specific queries.
- Access to all aspects of individual instances is required for the implementation of state rollback.

```
(defclass BRIDGE-PLAYER ()
  ((name  :initform "")
   (age   :initform 0))
  (:metaclass pclos-class))

;; Find all instances of CLOS class BRIDGE-PLAYER which are
;; named "Fred" or are older than 30:

(find-all <db-representative>
          'BRIDGE-PLAYER
          '(or
            (= name "Fred")
            (> age 30)))
```

Figure 4: PCLOS Queries Specified in Terms of CLOS Classes

All of these requirements impact exclusively the part of the MOP that is concerned with the definition and manipulation of classes. We therefore describe this part of the MOP in more detail. Please note again that we simplify some irrelevant aspects.

5.1 Metaclasses

In this section we examine in some detail how CLOS class metaobjects are created and manipulated by the MOP. The modifications that were necessary to satisfy PCLOS requirements will follow naturally from these descriptions and will be explained 'on the way'.

Recall that a CLOS class as defined by a CLOS programmer is implemented as an instance of a metaclass. There are five major aspects to a CLOS class:

- The class must be created and initialized.
- Forward references to yet to be created superclasses must eventually be resolved.
- The class must provide the means to create instances and to access its instances' slots.
- It must be possible to examine all aspects of the class programmatically.
- A class must sometimes be modified.

The following subsections will cover all but the last of these aspects[3]

Figure 5 shows the path we will take through the process of defining a CLOS class.

5.1.1 Class Creation

Class creation is triggered when the CLOS programmer evaluates a `defclass` statement[4]. CLOS begins its work with a call to `make-instance` on a metaclass. It returns a metaobject which will eventually hold in its state all the information that is relevant to the CLOS class being created. This includes super and subclasses, slots, a precedence list and more.

PCLOS-specific aspects:

The choice of which metaclass to instantiate for the creation of a class is the first point at which PCLOS enters the picture. Classes whose instances will have the potential of being persistent need to hold information about open databases, about which of its slots are persistent or transient and other relevant mate-

[3] CLOS does allow existing classes to be modified. The ORION system [9] handles class redefinition in the context of persistence. PCLOS does not currently support this.

[4] All operations described in this section may be initiated both programmatically and interactively. Without loss of generality we will use the interactive CLOS macros – such as `defclass` – in all examples and explanations.

Figure 5: Steps For Defining a Class

rial. Metaobjects that are CLOS classes therefore need some additional slots beyond those provided in standard CLOS class metaobjects.

Apart from these structural requirements, persistent classes will need modified and extended behavior.

PCLOS therefore defines a new metaclass which inherits from a standard CLOS metaclass but:

- Adds additional slots.
- Shadows some methods.
- Adds some methods.

When a programmer defines a CLOS class whose instances will have the potential of being made persistent, he specifies that his class is to be implemented not by the metaclass that generates standard CLOS classes, but by a new metaclass called `pclos-class`. Figure 6 shows a small example class hierarchy which we will use in the following material.

The only difference between a standard class definition and the one shown in the figure is the specification that the metaobject which will implement the `steam-ship` class is to be an instance of `pclos-class`.

5.1.2 Superclass Validation

After a class metaobject has been created, its future superclasses are examined to ensure that they are suitable to be parents of the class being defined. In particular, the system checks whether all superclasses are of the same metaclass as the one being defined. In our example, this process will check whether class `ship` also has metaclass `pclos-class`.

```
(defclass ship ()
  ((name :initform ""))
  (:metaclass pclos-class))

(defclass steam-ship (ship)
  ((color        :initform 'red)
   (registration :initform "Liberia"))
  (:metaclass pclos-class))
```

Figure 6: A CLOS Class Definition

The CLOS method `validate-superclasses` implements superclass validation. The default behavior of this method is to indicate incompatibility whenever the metaclass of any parent-to-be is not the same as that of the new class.

PCLOS-specific aspects:

PCLOS accepts the default behavior and therefore does not shadow this method. This means that PCLOS will behave just like the standard CLOS as far as class/superclass compatibility is concerned.

5.1.3 Local Slot Processing

After the superclasses are validated, the slot specifications in the `defclass` statement are checked for errors, and a metaobject of metaobject class `slot-descriptor` is created for each of the slots. Please refer to figures 7 and 8 for a graphical presentation of the following material.

Note that at this point only the slots directly specified for the class being defined are processed. We call these *local* slots. The merger with slots inherited from superclasses happens later.

Each `slot-descriptor` metaobject holds all relevant information about the slot it represents. Examples are the slot's name, any type limitations for values stored in it, or how it is to be initialized. The metaobject that implements the class will contain references to all these slot definition metaobjects.

The creation of a `slot-descriptor` happens in two stages: first, the system finds out which metaobject class the new descriptor is to be an instance of. Then the instance is created and initialized.

PCLOS-specific aspects:

PCLOS needs to interact with this default behavior in the following ways:

- Three slots must be added to the standard slot descriptor metaobjects.
- These new slots must be initialized.
- One new slot option must be added to the built-in CLOS slot options.

Slot descriptor metaobjects for persistent slots must be made to contain information on:

1. Whether the slot is persistent.
2. Whether its value is currently cached.
3. Whether its value has been saved since the last savepoint.

PCLOS defines a new metaobject class `persistent-slot-descriptor` which inherits from the CLOS default `slot-descriptor` and adds this information. Here is why the information is needed.

PCLOS allows individual slots to be transient, even if an instance as a whole is persistent. This is sometimes

Figure 7: Metaclasses

done to improve efficiency (e.g. loop-counters), for slots holding values that PCLOS cannot store in databases (e.g. Common Lisp compiled `function` objects) or for slots whose values make no sense across sessions (e.g. a current window object).

Programmers must therefore be able to specify transience for slots. This is done through a new slot option called `:transient`. CLOS slot options are declarations which may be specified for individual slots when a class is defined. Examples for built-in options are '`:allocation`' and '`:type`'. The first allows the programmer to specify whether the associated slot is to be shared among all instances of the class that contains the slot, or whether each instance is to have a private copy. The `:type` option restricts the values that may be stored in the associated slot.

Figure 9 shows a new version of our `steam-ship` class that includes a transient slot.

In order to extend slot option error checking and slot initialization to include the new `:transient` option, PCLOS specializes CLOS' `shared-initialize` method to `persistent-slot-descriptor`. This new method takes care of PCLOS-related items but leaves the remainder of the error checking and initialization to the standard, built-in mechanisms. This is done by explicitly invoking the parent method which handles these affairs for regular, default slot descriptors.

5.1.4 Class Initialization Arguments

After the locally specified slots have been processed as described above, class *initargs* are checked for errors. `Initargs` let the designer of a class specify which

Figure 8: Slot Descriptors

```
(defclass steam-ship (ship)
  ((color         :initform 'red)
   (registration  :initform "Liberia")
   (current-speed :initform 0 :transient T))
  (:metaclass pclos-class))
```

Figure 9: Example of a Transient Slot

slots may be initialized 'in-line' as part of the call to `make-instance` on the class being defined.

PCLOS does not need to modify this built-in default mechanism.

5.1.5 Computing the Class Precedence List

Class inheritance can lead to conflicts among inherited slots and methods which must be resolved before a new class is put to use. This is done by the method `compute-class-precedence-list` which is called with the class metaobject being defined. This method returns a list of class metaobjects in most-specific-first order. It flattens a multiple inheritance hierarchy into a linear ordering imposed for the purpose of resolving name conflicts and selecting methods at run-time.

PCLOS-specific aspects:

Instances of persistence-ready classes must respond to many persistence-related messages. Examples are cache, write-back[5] or unprotect[6]. Typically, such operations would be inherited from some appropriate superclass that programmers explicitly included in the inheritance specification of their classes. PCLOS avoids the necessity of listing such a superclass with every class specification by ensuring that this superclass automatically becomes the maximal element in the class precedence list.

5.1.6 Final Slot Analysis

Near the completion of a class definition, all conflicts between slot specifications must be resolved in accordance with the previously computed class precedence list. This is done by a method which examines slot aspects, such as typing, in slots that are defined in several classes. The method then produces `effective-slot-descriptor` metaobjects for each locally defined slot and for the slots inherited from superclasses.

PCLOS does not need to modify this default mechanism.

After all the above steps have been completed, a newly defined class is said to be *finalized.* Finalized classes are ready to be instantiated.

This review of the class definition process and its modification on behalf of PCLOS have shown that all persistence-related needs have been met by simple, judicious shadowing of metalevel methods.

[5] This writes an instance's slot values to the underlying database without uncaching it.

[6] This causes the instance to no longer be persistent.

5.2 Inspecting CLOS Language Elements

We have seen how CLOS internal data structures and processes can be modified. To attain any degree of persistence transparency, we also need the ability to inspect many normally unavailable language elements: When preparing a database for receiving objects to be stored, PCLOS must, for instance, examine details of the instances and their classes: the list of a class' slots must be available to determine how to generate an appropriate schema. For the same reason PCLOS must find out whether a slot is `:class` or `:instance` allocated, whether it is typed and whether it is persistent.

The process of translating queries phrased in CLOS-like expressions to queries understood by the underlying databases also requires language elements to be inspectable.

When objects are cached into main memory, the rollback guarantees of PCLOS imply that slot values must be copied before they are modified for the first time after a savepoint. This implies that slots must have an attribute that indicates whether the slot is still 'clean' and must therefore be saved before an impending update, or whether it is already 'dirty' and has therefore been saved earlier.

An important implication of the CLOS design is that all this information may be kept and retrieved in an orderly and modular fashion. The expression

```
(find-class <symbol>)
```

returns the metaobject that implements the class with the specified name `<symbol>`. With that class metaobject in hand, all necessary information may be retrieved through 'officially sanctioned' means: a list of all effective slot descriptors may, for instance, be obtained through

```
(class-slots <class-metaobject>)
```

The allocation specification of a slot may be obtained through

```
(slot-descriptor-allocation
   <slot-descriptor-metaobject>)
```

5.3 Run-time Operation

There are three sets of activities that relate to run-time operations and that affect PCLOS:

- Instance allocation and layout.
- Slot access.
- Slot access optimization.

In CLOS the low-level mechanics of reading and writing slots are implemented by the methods `slot-value-using-class` and `setf-slot-value-using-class` respectively. These methods are just below the programmer-visible `slot-value` method and are the only means to read or write slots, except for access optimizations which will be considered later in this section.

PCLOS-specific aspects:

Figure 10 shows the details of how PCLOS reads a slot value. The terms *C-slots* and *I-slots* in the figure refer to `:class` and `:instance` allocated slots respectively.

This algorithm is implemented in a PCLOS-specialized `slot-value-using-class` method. At this level, writing a slot value is symmetric to slot reading that is shown in the figure. If an instance is not persistent, reading immediately proceeds along the standard CLOS slot access algorithm. Access to both non-persistent instance-private and globally visible slots are roughly on the order of a vector element read.

Slot access to non-persistent instances therefore suffers the cost of one conditional branch beyond the default CLOS implementation. The information necessary to evaluate that conditional is an in-line vector element fetch.

A slot value of a persistent instance is memory-resident under several conditions:

- The slot in question is transient.
- The entire instance is cached.
- The slot is individually cached.

If the slot value is memory-resident, the access continues as for non-persistent instances. Otherwise a database query – or update in the case of slot value modification – is neccessary to complete the slot access.

Note that some databases themselves cache information, so a query may still be reasonably fast, although of course slower than a vector fetch. PCLOS tries to take advantage of query compilation on databases that support such optimizations.

CLOS itself provides a framework for slot access optimization. This enables metaobject programmers to provide in-line, highly optimized, albeit non-portable slot access mechanisms based on their knowledge about the system implementation. PCLOS cannot take advantage of this capability, because the location of slot values is generally not known at compile time.

Instance allocation, finally, is accomplished by the method `allocate-instance` on `class` metaobjects. This method allocates space for the slot values of new instances and any other slot-related information. Note that Lisp's automatic memory management makes allocation much easier than it would be for a language in

Figure 10: Reading a Slot Value

which a programmer himself is responsible for allocating and managing memory space. Since slot values may vary in size, such data management will be unpleasant for those languages, unless they are very strictly typed.

Instance allocation is of interest to PCLOS for two reasons:

- The structure of the allocated space must accommodate all of an instance when it is cached.
- PCLOS needs to maintain some per-instance administrative information.

Once a PCLOS instance is created either explicitly by the CLOS call to `make-instance`, or implicitly because a database query brought the instance into the current environment, a *husk* is created which is the placeholder for the instance in the current session. When slot values of an instance reside in a database, the husk contains no useful data, but it looks like a valid CLOS instance. This is important for making memory-resident references to the instance work properly[7]. When an instance is cached, the slot values will be placed in the husk.

For each memory-resident instance PCLOS must therefore maintain information, such as which database the instance is associated with, whether the instance has been cached or whether it is *dirty*. An instance is considered dirty if its slot values or any of its administrative information have been modified since it was cached.

PCLOS allocates extra space for this information whenever it allocates an instance. This is simply accomplished by specializing the CLOS `allocate-instance`

[7] When an object reference is stored in a database, the memory pointer is replaced by a special, database-dependent encoding. Upon retrieval of the reference, a correspondence is established between the encoded reference and the current location of the referenced, memory-resident object husk. If such a husk does not yet exist, it is created.

method for the `pclos-class` metaclass. An alternative approach would have been to add 'hidden slots' which were 'censored' out of all the public means for describing and manipulating instances. In the final analysis, the two approaches are probably equivalent[8].

6 Related Work

Two domains of related work suggest themselves as material supplementary to this paper. One is CLOS's architectural base in the metaobject protocol approach, the other is the area of object persistence.

6.1 Language-Related Work

The idea of making seemingly fundamental components of systems in reality be elements of a meta-level 'world' has been explored in various earlier systems.

Like CLOS, Smalltalk [10] includes the notion of metaclasses. But the concept, though equal in name, is quite different in the two languages: Each Smalltalk class is an instance of exactly one metaclass which may only have that one class as its instance. A class thereby acts like 'regular', program-level objects in the sense that it responds to messages whose effects are determined by its (meta)class. In particular, the metaclass controls the initialization of class variables. It also manufactures the class' instances. But in contrast to CLOS, the programmer cannot modify metaclasses and use object-oriented programming at the metalevel to produce special effects.

ObjVlisp [11], which is very similar to CLOS [12], has worked on introducing a full metalevel class mechanism into Smalltalk-80 [13]. This has led to a kind of 'metaclass workbench' called *Classtalk* which helps with the construction of metaclass libraries and provides a metaclass browser.

[8] But I like mine better...

An interesting angle to metalevel architectures is added by [14] which shows how the principle can be used in the construction of operating systems.

There is a rapidly accumulating body of literature about CLOS and its uses. Another report on the use of the metaobject protocol can be found in [15]. The authors illustrate how the MOP could be used to implement concurrency control for shared objects. Their second example outlines how persistence could be implemented through metalevel manipulations. This is in several ways similar to PCLOS, except that their example targets a single data store and does not concern itself with transparency. Their final example explains how graphic objects could be implemented with the CLOS metaobject protocol.

The first and second "CLOS Users and Implementors Workshops" of 1988 and 1989 are good sources for information on a wide spectrum of CLOS aspects.

6.2 Persistence-Related Work

A broad survey of language persistence can be found in [16]. An earlier paper about PCLOS [2] also provides a long list of references. In this paper we will limit ourselves to pointing out some research reports that specifically exemplify different techniques for introducing persistence into programming languages. The main techniques are full integration by modifying the language, construction of persistence-related mechanisms within the regular boundaries of the language, and the invention of a new language.

PCLOS and [17] are examples of full language integration attempts. They rely on the metaobject protocol facility for their implementation. Avalon [18, 19] is radically different in its approach to transparency and in its C++ affinity, but it also attempts to introduce persistence by modifying a language. This is done through preprocessing. Reference [20] in particular, addresses the issue of language integration.

PS-Algol [21] was an early effort in providing language persistence. The implementation of persistence was done by functional extensions to an Algol-like language. It therefore stayed 'within' the language. In [22] the authors added guidelines for the use of typing to introduce persistence into languages. Reference [23], finally, examines how procedures may be treated as persistent data objects, a facility not addressed by PCLOS.

In some ways similar to PS-Algol is the Coral 3 system [24] which provides persistence for Smalltalk. Although the respective languages are very different, the approach of both is to develop some model of persistence and to implement it without modifying the language. In the case of Coral 3, an added touch of 'go with what's at hand' has storage be provided by the file system instead of using a database.

The GemStone system [25] introduces a new language, OPAL, to deal with persistence. This can be used as the sole application language, or it can be used in conjunction with Smalltalk [26] or C. The advantage of this approach is that data processing and storage can be tuned to each other. The disadvantage is that a programmer must learn a new language to enjoy the full power of the system.[9].

7 Conclusion

We began with the observation that the tension between the goals of system portability and system flexibility may be reconciled by standardizing not only the system, but also the mechanisms for introducing – possibly fundamental – modifications. The portability of a standardized piece of machinery is thereby extended to cover multiple, different variations of that machinery.

We then put forth the suggestion that the design of the object-oriented Common Lisp Object System (CLOS) is an example for the technique of organizing the static elements and the dynamic processes of a system in such a way that significant modifications can be accommodated through well-planned, pervasive degrees of freedom.

The remainder of the paper was dedicated to supporting this suggestion by showing how the existing PCLOS system adds persistence to CLOS without modifying any CLOS implementation code.

Such major modification would normally require significant incisions into the language implementation:

- All slot access must be intercepted, for both reads and writes.
- Close to all aspects of the fundamental language elements must be extensively self-describing.
- The representations of basic elements, such as slots and instances must be extended.

We showed in some detail that CLOS can satisfy all of these requirements through well-known object-oriented techniques applied at the CLOS metaobject level: Specialization of existing classes, shadowing of inherited methods and the addition of new methods.

PCLOS has shown that the CLOS metaobject protocol mechanism for implementing a language is very powerful. As the CLOS metaobject protocol continues to evolve and CLOS is made available on a growing number of machines, we can look forward to other interesting applications. We can also hope that these applications will generate insights and experience that will enable us to generalize the metaobject protocol idea to benefit system components other than programming languages.

[9] There are Smalltalk and C interfaces to Opal which can ease this problem.

8 Acknowledgments

Credit goes to the Xerox Corporation for making the PCL implementation developed at Xerox PARC available to the research community. This availability has enabled CLOS to benefit from the suggestions of a broad base of researchers everywhere.

In the early days, Jim Kempf and Roy D'Souza ported CLOS to our workstations and made many improvements there. While others then had problems with their ports and the maintenance of CLOS, we were able to get an early start thanks to their efforts.

My local CLOS co-users who helped to understand, test and debug the language included Mike Creech, Cathy Fletcher, Dennis Freeze, and Craig Zarmer.

Users of PCLOS, finally, were of particular value to me and included the colleagues above, as well as Lucy Berlin, Vicki O'Day and Bob Leichner. Charles Hoch and Peter Lyngbaek of the Iris database group at HP-Labs, finally, helped me to understand and use their object-oriented database which can be accessed through PCLOS.

Warren Harris suggested many important improvements to an early draft of this paper.

References

[1] Andreas Paepcke. PCLOS: A Flexible Implementation of CLOS Persistence. In S. Gjessing and K. Nygaard, editors, *Proceedings of the European Conference on Object-Oriented Programming*, pages 374–389. Lecture Notes in Computer Science, Springer Verlag, 1988.

[2] Andreas Paepcke. PCLOS: A Critical Review. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, 1989.

[3] Daniel G. Bobrow, Linda DeMichiel, Richard P. Gabriel, Gregor Kiczales, David Moon, and Sonya Keene. The Common Lisp Object System specification: Chapters 1 and 2. Technical Report 88-002R, X3J13 standards committee document, 1988.

[4] Sonya E. Keene. *Object-Oriented Programming in Common Lisp.* Addison-Wesley Publishing Company, 1989.

[5] Guy L. Steele Jr. *Common Lisp: The Language.* Digital Press, second edition, 1990.

[6] Gregor Kiczales and Daniel G. Bobrow. The Common Lisp Object System specification: Metaobject protocol. Technical Report 88-003, X3J13 standards committee document, 1988.

[7] Daniel G. Bobrow and Gregor Kiczales. The Common Lisp Object Ssystem metaobject kernel: A status report. In *Conference on Lisp and Functional Programming*, 1988.

[8] D. Fishman et al. Iris: An object-oriented database management system. *ACM Transactions on Office Information Systems*, 5(1):48–69, April 1987.

[9] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In Umeshwar Dayal and Irv Traiger, editors, *Proceedings of the ACM Special Interest Group on Management of Data.* Association of Computing Machinery, 1987.

[10] Adele Goldberg and David Robinson. *Smalltalk-80: The Language and Implementation.* Addison Wesley, 1983.

[11] Pierre Cointe. Metaclasses are first class: The ObjVlisp model. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications.* Association of Computing Machinery, 1987.

[12] P. Cointe and N. Graube. Programming with metaclasses in clos. In *Proceedings of the First CLOS Users and Implementors Workshop*, 1988.

[13] Jean-Pierre Briot and Pierre Cointe. Programming with explicit metaclasses in Smalltalk-80. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, 1989.

[14] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. A reflective architecture for an object-oriented distributed operating system. In *Proceedings of the European Conference on Object-Oriented Programming*, 1989.

[15] Guiseppe Attardi, Cinzia Bonini, Maria Rosaria Boscotrecase, Tito Flagella, and Mauro Gaspari. Metalevel programming in CLOS. In *Proceedings of the European Conference on Object-Oriented Programming*, 1989.

[16] Malcolm P. Atkinson and O. Peter Buneman. Types and persistence in database programming languages. *Computing Surveys*, 19(2):105–189, June 1987.

[17] Lawrence A. Rowe. A shared object hierarchy. In Klaus Dittrich and Umeshwar Dayal, editors, *Proceedings of the International Workshop on Object-Oriented Database Systems.* Association of Computing Machinery, 1986.

[18] Jeannette M. Wing, Maurice Herlihy, Steward Clamen, David Detlefs, Karen Kietzke, Richard Lerner, and Su-Yuen Ling. The Avalon/C++ programming language. Technical Report CMU-CS-88-209, Carnegie Mellon University, December 1988.

[19] Richard A. Lerner. Reliable servers: Design and implementation in Avalon/C++. In *Int'l Symp. on Databases in Parallel and Distributed Systems.* IEEE, 1988.

[20] David Detlefs, Maurice Herlihy, Karen Kietzke, and Jeannette Wing. Avalon/C++: C++ extensions for transaction-based programming. In *USENIX C++ Workshop*, 1987.

[21] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, 1983.

[22] M.P. Atkinson and R. Morrison. Integrated persistent programming systems. In B.D. Shriver, editor, *Proceedings of the 19th Annual Hawaii Conference on System Sciences*, pages 842–854, 1986. Vol. IIA, Software.

[23] M.P. Atkinson and R. Morrison. Procedures as persistent data objects. *ACM Transactions on Programming Languages and Systems*, 7(4):539–559, October 1983.

[24] Thomas Merrow and Jane Laursen. A pragmatic system for shared persistent objects. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications.* Association of Computing Machinery, 1987.

[25] David Maier, Jacob Stein, Allen Otis, and Alan Purdy. Development of an object-oriented DBMS. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications.* Association of Computing Machinery, 1986.

[26] G. Copeland and D. Maier. Making Smalltalk a database system. In *Proceedings of the ACM/SIGMOD International Conference on the Management of Data*, 1984.