A Survey of Current CLOS MOP Implementations

Raymond de Lacaze
Artificial Intelligence Center
SRI International
333 Ravenswood Ave.
Menlo Park, CA 9402
delacaze@ai.sri.com

Tim Bradshaw
Cley Limited
6 East Hermitage Place
Edinburgh EH6 8AA
UK
tfb@cley.com

April 20, 2000

Abstract

This paper surveys implementations of the CLOS MOP in some of the more popular implementations of Common Lisp. First a brief overview of the CLOS MOP is given. A number of Common Lisp implementations are then examined independently in order to assess which aspects of the CLOS MOP they provide. A comparison is made between the various implementations. Particular attention is given to what portion of the CLOS MOP is implemented and how faithful the implementation is to the metaobject protocol described in the *The Art of the Metaobject Protocol*[1]. A summary of the overall availability of the CLOS MOP is then given. Finally a proposed categorization of the functionality of the CLOS MOP is described. This work is related to an ongoing effort being done by the ANSI Common Lisp J13 committee.

1 Introduction

A metaobject protocol is an interface to elements of a language that are normally hidden from users of the language. Providing such an interface allows users of the language to tailor the language to their own needs and in such a way provides a more powerful and flexible language. The CLOS metaobject protocol provides an interface to program elements such as classes and methods, thereby allowing users to control aspects of the language such as how instances of a class are created or how method dispatching works. Although the CLOS metaobject protocol has not yet been standardized, many implementations already provide a metaobject protocol based on the protocol described in *The Art of the Metaobject Protocol*[1], henceforth referred to as AMOP.

2 The CLOS MOP

As described in AMOP[1], a CLOS implementation (and implementations of other languages for that matter) can be seen as consisting of two sets of components: the on-stage components and the backstage components. The on-stage components are those aspects of the implementation that are available to users of the language and are typically the documented entities. The backstage components are the things that an implementation needs to provide in order to support the on-stage language.

Another useful decomposition made by CLOS implementations is the three-layered division of the defining forms (i.e. of *defclass*, *defmethod* and *defgeneric*). These are the macro-expansion-layer, the glue layer and the lowest layer. The macro-expansion layer consists of on-stage entities while the lowest layer consists of backstage entities. The glue layer can be seen as piecing together these two layers. It does so by mapping the names used in the macro-expansion layer (e.g. class names) to the objects that represent them in the lowest layer (e.g. class objects). For example *find-class* is a component of the glue layer which takes a class name as an argument and returns a class object.

Languages typically do not expose their backstage mechanisms to the user of the on-stage language. This results in language which is inherently fairly rigid – it occupies a single point in the space of possible designs. Even if the design is good, it is unlikely to be adequate for all users. Lisp has traditionally been extraordinarily flexible in the way it can provide a range of possible languages rather than a single language. Making Lisp source code be Lisp data, and providing a macro language which is Lisp itself has led to a style of problem solving where the language is tailored to the problem at hand, a style which is almost unique to Lisp. So it is very much in the Lisp tradition to expose some of the backstage mechanisms to allow yet more flexibility.

Exposing some of the backstage mechanisms should allow the user of the language to further customise the language to the problem at hand, rather than crushing the problem into the fixed, rigid framework provided by the language. As well as the default on-stage language provided 'out of the box', variant on-stage object-oriented languages can be designed which fit the problem space, and which then allow people to concentrate on solving the problem rather than fighting the language.

In an object-oriented language, like CLOS, the behaviour of objects is largely determined by their classes or types. To expose the backstage behaviour is then naturally done by defining the behaviour of CLOS itself in terms of CLOS objects, and allowing the user to select which objects are used, within reason, to define the behaviour of the program. So, in CLOS, a class is itself an object which is an instance of a class, and this class determines the behavious of the on-stage CLOS objects. In CLOS, classes are known as metaobjects, and the classes of these metaobjects are known as metaclasses. Exposing what these metaobjects and metaclasses are, and defining the protocols they follow and how they can be changed, then allows the user to tailor the on-stage language. The protocol followed by these metaobjects in CLOS is the CLOS Metaobject Protocol.

There is obviously a circularity involved when defining CLOS in terms of itself this way: metaclasses are themselves metaobjects, with their own meta-metaclasses and so on. If the backstage protocol is exposed then these circularities need to be dealt with,

and this is done by special-casing behaviour for certain known metaobjects to prevent infinite regress. Figure 1 shows the metaclass hierarchy for CLOS as defined in AMOP.

The essence of a metaobject protocol is to provide an interface to these metaobjects, a set of methods and functions that allow users not the only the ability to access these metaobjects, but perhaps more importantly, the flexibility to control the creation and manipulation of both objects and metaobjects. This allows users to control the behavior of the CLOS language itself. For example users can control the class initialization protocol and even the generic function invocation protocol. Such an interface effectively moves these metaobjects on-stage. It should be pointed out that this level of control cannot be achieved by simply redefining existing CLOS functions as this would potentially break any existing CLOS program, but rather by defining new metaclasses and writing new methods that specialize generic functions in the metaobject protocol. A well designed metaobject protocol does so in a way that optimizes the control users have over the language and yet does not compromise the control implementors need over the language in order to ensure that efficient implementation is reasonably feasible. Achieving this delicate balance was an important goal of the authors of AMOP[1], and if a MOP is to be standardised it is very important that it should meet these requirements.

The authors of AMOP[1] did not intend their CLOS metaobject protocol to be the suggested standard. However, most implementations have used it as the basis for their implementation of the CLOS MOP and it has become the *de facto* standard. Though the CLOS MOP has not been standardized, the current ANSI standard for Common Lisp includes CLOS and does provide access to a set of metaobjects, the standard metaobjects, and a set of functions and methods to manipulate these metaobjects, which strictly speaking should be considered part of the CLOS MOP. However most of these were not intended to be manipulated by users of the language, but rather serve to specify the behavior of the Object System in Common Lisp to implementors of the language. The standard metaobject classes are *standard-class*, *standard-method*, *standard-generic-function* and *standard-slot-definition*.

Perhaps one aspect of the design of the CLOS MOP which is not complete is the structure of the *method-combination* metaclass. As it stands the *de facto* standard does not specify the structure of this metaobject, although the ANSI CLOS standard does document the *define-method-combination* macro, which can be used to create new types of method combinations. The default CLOS method combination is referred to as standard method combination, and it seems reasonable that the CLOS MOP should at least provide a *standard-method-combination* metaclass as part of the standard metaobjects and appropriate reader methods. Many implementations actually implement such a class. This is perhaps one area in which the CLOS MOP could be extended.

In the following sections, we survey the availability of the CLOS Metaobject Protocol in several popular Common Lisp implementations.

3 Allegro Common Lisp

Franz Inc.'s Allegro Common Lisp version 5.0.1, available from www.franz.com, provides a comprehensive implementation of the full CLOS MOP. All of the metaclasses

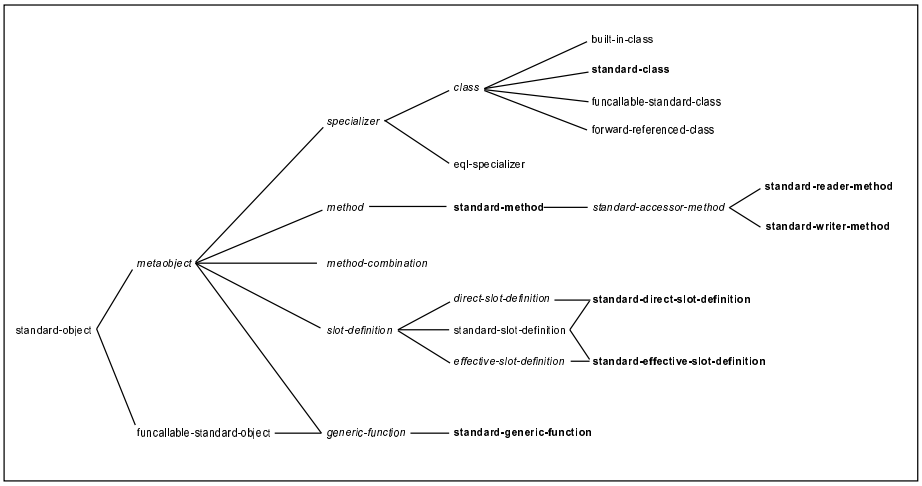

Figure 1: The CLOS MOP Metaclass Hierarchy. Classes in bold face represent the standard metaobject classes and classes in italic denote abstract classes not intended to be instantiated.

and generic functions described in AMOP[1] are implemented in this version of Allegro. The Allegro CL User Guide[2] provides complete documentation of these MOP entities as well as numerous examples allowing users to effectively use the Allegro MOP implementation. Although this is not listed as one of the standard CLOS metaobject classes, Allegro Common Lisp does implement a *standard-method-combination* metaclass.

4 CLISP

Although CLISP Common Lisp, written by Bruno Haible and Michael Stoll, and available from clisp.sourceforge.net provides some of the CLOS MOP, many of of the entities described in the the *de facto* standard are missing or else appear under a different name. Most of the class metaobject reader methods are provided. However none of the *method*, *generic-function* and *slot-definition* reader methods are supported. It is not clear from the release notes[3], which describe some of the unsupported aspects of CLOS, if the implementors of CLISP plan to provide a more standard and complete implementation of the CLOS MOP as defined by AMOP[1] in future releases.

5 Corman Lisp

Roger Corman's Corman Lisp version 1.3, available from www.corman.net, provides about half of the CLOS MOP entities. Those components of the MOP which relate to *slot-definition* metaobjects and *eql-specializer* metaobjects are not supported. In addition the classes and functions related to the following MOP concepts are also absent:

Abs	MetaClass Name	Allegro	CLISP	Corman	Lispworks	MCL
	built-in-class	Yes	Yes	Yes	Yes	Yes
(*)	class	Yes	Yes	Yes	Yes	Yes
(*)	direct-slot-definition	Yes	No	No	Yes	Yes
(*)	effective-slot-definition	Yes	No	No	Yes	Yes
	eql-specializer	Yes	No	No	No	No
	forward-referenced-class	Yes	No	No	Yes	No
	funcallable-standard-class	Yes	No	No	Yes	Yes
	funcallable-standard-object	Yes	No	No	No	Yes
(*)	generic-function	Yes	Yes	Yes	Yes	Yes
(*)	metaobject	Yes	No	No	Yes	Yes
(*)	method	Yes	Yes	Yes	Yes	Yes
(*)	method-combination	Yes	No	Yes	Yes	Yes
(*)	slot-definition	Yes	Yes	No	Yes	No
(*)	specializer	Yes	No	Yes	Yes	Yes
(*)	standard-accessor-method	Yes	No	No	Yes	Yes
	standard-class	Yes	Yes	Yes	Yes	Yes
	standard-direct-slot-definition	Yes	No	No	Yes	No
	standard-effective-slot-definition	Yes	No	No	Yes	No
	standard-generic-function	Yes	Yes	Yes	Yes	Yes
	standard-method	Yes	Yes	Yes	Yes	Yes
	standard-object	Yes	Yes	Yes	Yes	Yes
	standard-reader-method	Yes	No	No	No	Yes
(*)	standard-slot-definition	Yes	Yes	No	Yes	No
	standard-writer-method	Yes	No	No	No	Yes

Figure 2: Availability of MOP classes in several CLOS implementations. The classes marked with (*) are abstract classes that are not intented to be instantiated.

direct subclasses, direct methods, method combination, dependents, reader methods, writer methods and funcallable objects. The metaobject protocol provided in Corman Lisp is based on *Closette*. This is a metaobject protocol described in AMOP[1], which is simpler than the full CLOS MOP and was used by the authors AMOP[1] as a means of stepping the reader through the design and implementation of a CLOS metaobject protocol without having to immediatedly address issues pertaining to the provision of a MOP for the full CLOS language. The Corman Lisp User Guide and Reference[5] outlines which aspects of the CLOS and the MOP are currently unsupported, and indicates that work is in progress for providing the full ANSI CLOS and the *de facto* MOP. It should also be pointed out that Corman Lisp provides the most commonly used components of the CLOS MOP, while the unsupported ones are in some sense the more advanced aspects of the CLOS MOP.

6 Lispworks

Harlequin's Lispworks 4.01, now available from Xanalys at www.xanalysis.com, provides nearly the complete CLOS MOP. There are only a handful of entities missing. In particular the *eql-specializer* metaclass, the *funcallable-standard-object* metaclass and the *standard-reader-method* and *standard-writer-method* metaclasses are not pro-

Type	Function Name	Allegro	CLISP	Corman	Lispworks	MCL
GF	accessor-method-slot-definition	Yes	No	No	No	Yes
GF	add-dependent	Yes	No	No	Yes	Yes
GF	add-direct-method	Yes	No	No	Yes	Yes
GF	add-direct-subclass	Yes	No	No	Yes	Yes
GF	add-method	Yes	Yes	Yes	Yes	Yes
GF	allocate-instance	Yes	Yes	Yes	Yes	Yes
GF	class-default-initargs	Yes	Yes	No	Yes	No
GF	class-direct-default-initargs	Yes	Yes	No	Yes	No
GF	class-direct-slots	Yes	Yes	Yes	Yes	Yes
GF	class-direct-subclasses	Yes	No	Yes	Yes	Yes
GF	class-direct-superclasses	Yes	Yes	Yes	Yes	Yes
GF	class-finalized-p	Yes	No	No	Yes	Yes
GF	class-name	Yes	Yes	Yes	Yes	Yes
GF	class-precedence-list	Yes	Yes	Yes	Yes	Yes
GF	class-prototype	Yes	No	No	Yes	Yes
GF	class-slots	Yes	Yes	Yes	Yes	Yes
GF	compute-applicable-methods	Yes	Yes	Yes	Yes	Yes
GF	compute-applicable-methods-using-classes	Yes	No	Yes	No	Yes
GF	compute-class-precedence-list	Yes	No	Yes	Yes	Yes
GF	compute-default-initargs	Yes	No	No	Yes	Yes
GF	compute-discriminating-function	Yes	No	Yes	Yes	Yes
GF	compute-effective-method	Yes	Yes	No	Yes	Yes
GF	compute-effective-slot-definition	Yes	No	Yes	Yes	Yes
GF	compute-slots	Yes	No	Yes	Yes	Yes

Figure 3: Availability of MOP functions in several CLOS implementations.

vided. The generic functions that manipulate metaobjects of these metaclasses are also unsupported. These include the generic functions *eql-specializer-object*, *intern-eql-specializer*, *compute-applicable-methods-using-classes*, *reader-method-class* and *writer-method-class*. Lispworks also provides a *standard-method-combination* class.

7 MCL

MCL 2.0, available from Digitool at www.digitool.com, provides most of the MOP entities. It implements the CLOS language as described in CLTL2. The documentation is wonderfully explicit about which components of the CLOS MOP are not provided in this release. The unsupported metaobject classes primarily are those relating to the slots. This includes slot-definition, *standard-slot-definition*, *standard-direct-slot-definition* and *standard-effective-slot-definition*. The metaclasses *eql-specializer* and *forward-referenced-class* are also unavailable in this version of MCL. It is noted in the documentation of MCL that when an unknown class is encountered at compile, an instance of the class *ccl::compile-time-class* is created instead, and that this is likely to change in future versions of MCL. It is also noted that future versions will completely support the *slot-definition* metaclass. The generic functions in the AMOP[1] that are not available in this release include most of the ones pertaining to *slot-definition* metaobjects, the generic-function reader methods *generic-function-*

Type	Function Name	Allegro	CLISP	Corman	Lispworks	MCL
GF	direct-slot-definition-class	Yes	No	No	Yes	Yes
GF	effective-slot-definition-class	Yes	No	No	Yes	Yes
Fn	ensure-class	Yes	Yes	Yes	Yes	Yes
GF	ensure-class-using-class	Yes	No	No	Yes	Yes
Fn	ensure-generic-function	Yes	No	Yes	Yes	Yes
GF	ensure-generic-function-using-class	Yes	No	No	Yes	Yes
Fn	eql-specializer-object	Yes	No	No	No	Yes
Fn	extract-lambda-list	Yes	No	Yes	Yes	Yes
Fn	extract-specializer-names	Yes	No	No	Yes	Yes
GF	finalize-inheritance	Yes	No	Yes	Yes	Yes
GF	find-method-combination	Yes	No	No	Yes	Yes
Fn	funcallable-standard-instance-access	Yes	No	No	No	Yes
GF	generic-function-argument-precedence-order	Yes	No	No	Yes	No
GF	generic-function-declarations	Yes	No	No	Yes	No
GF	generic-function-lambda-list	Yes	No	Yes	Yes	No
GF	generic-function-method-class	Yes	No	Yes	Yes	Yes
GF	generic-function-method-combination	Yes	No	No	Yes	Yes
GF	generic-function-methods	Yes	No	Yes	Yes	Yes
GF	generic-function-name	Yes	No	Yes	Yes	Yes
Fn	intern-eql-specializer	Yes	No	No	No	Yes
GF	make-instance	Yes	Yes	Yes	Yes	Yes
GF	make-method-lambda	Yes	No	No	Yes	Yes
GF	map-dependents	Yes	No	No	Yes	Yes
GF	method-function	Yes	No	Yes	Yes	Yes
GF	method-generic-function	Yes	No	Yes	Yes	Yes
GF	method-lambda-list	Yes	No	Yes	Yes	No
GF	method-specializers	Yes	No	Yes	Yes	Yes
GF	method-qualifiers	Yes	Yes	Yes	Yes	Yes

Figure 4: Availability of MOP functions in several CLOS implementations (cont).

declarations, *generic-function-lambda-list* and *generic-function-argument-precedence-order*. Finally the generic functions *class-default-initargs*, *class-direct-default-initargs* and *method-lambda-list* are also unsupported. MCL does however provide a *standard-method-combination* class, which as already noted, is lacking from the *de facto* MOP standard.

8 Other Lisps

There are several other popular implementations of Common Lisp that we plan on examining. At the time this paper was written, we did not have easy access to these implementations:

- Liquid Common Lisp (formerly Lucid Common Lisp) is also available from Xanalysis at www.xanalysis.com.
- CMU Common Lisp was originally done at CMU and is now a community project which lives at www.cons.org.

Type	Function Name	Allegro	CLISP	Corman	Lispworks	MCL
GF	reader-method-class	Yes	No	No	No	Yes
GF	remove-dependent	Yes	No	No	Yes	Yes
GF	remove-direct-method	Yes	No	No	Yes	Yes
GF	remove-direct-subclass	Yes	No	No	Yes	Yes
GF	remove-method	Yes	Yes	Yes	Yes	Yes
Fn	set-funcallable-instance-function	Yes	No	No	Yes	Yes
Fn	(setf class-name)	Yes	Yes	Yes	Yes	Yes
Fn	(setf generic-function-name)	Yes	No	Yes	Yes	Yes
GF	(setf slot-value-using-class)	Yes	No	Yes	Yes	Yes
GF	slot-boundp-using-class	Yes	No	Yes	Yes	No
GF	slot-definition-allocation	Yes	No	Yes	Yes	No
GF	slot-definition-initargs	Yes	No	Yes	Yes	No
GF	slot-definition-initform	Yes	No	Yes	Yes	No
GF	slot-definition-initfunction	Yes	No	Yes	Yes	No
GF	slot-definition-name	Yes	No	Yes	Yes	No
GF	slot-definition-readers	Yes	No	Yes	Yes	No
GF	slot-definition-type	Yes	No	No	Yes	No
GF	slot-definition-writers	Yes	No	Yes	Yes	No
GF	slot-definition-location	Yes	No	No	Yes	Yes
GF	slot-makunbound-using-class	Yes	No	Yes	Yes	No
GF	slot-value-using-class	Yes	No	Yes	Yes	No
GF	specializer-direct-generic-functions	Yes	No	No	Yes	Yes
GF	specializer-direct-methods	Yes	No	No	Yes	Yes
Fn	standard-instance-access	Yes	No	No	Yes	Yes
GF	update-dependent	Yes	No	No	Yes	Yes
GF	validate-superclass	Yes	No	No	Yes	Yes
GF	writer-method-class	Yes	No	No	No	Yes

Figure 5: Availability of MOP functions in several CLOS implementations (cont).

- Symbolics Inc,'s Genera is an implementation of major historical importance.

We believe that LCL provides a comprehensive implementation of the MOP. Genera provides some MOP facilities but its implementation of CLOS varies at some levels. CMUCL's CLOS is a version PCL, which was the original testbed implementation of CLOS, and has a MOP, but one which does not always agree with the one described in AMOP.

9 Comparison

Among the three commercial implementations examined, Allegro CL and LispWorks provide the complete MOP (though LispWorks lacks just a few entities). MCL is not too far off, and essentially just needs to provide the *slot-definition* and *eql-specializer* related components. The two free lisps examined, CLISP and Corman Lisp are missing larger portions of the MOP and it would require a more significant effort on the implementors of these Common Lisps to provide the full MOP. It is however encouraging that they have implemented a substantial portion of the MOP and certainly impressive given that they are freely available implementations of Common Lisp. It should be

noted that these comments are not intented as an evaluation of any of these implementations but are rather geared towards providing the Common Lisp J13 Committee with a better understanding of the impact and burden that fully standardizing or enhancing the CLOS MOP would have on implementors of existing Common Lisp implementations.

Figure 2 summarizes the CLOS MOP classes which are available in each of the implementations discussed. The *slot-definition* metaobject class and its descendants, and the *eql-specializer* metaclasses are the most commonly unsupported aspects of the MOP in the implementations examined. Figures 3, 4 and 5 show exactly which CLOS MOP functions are available in each of the implementations discussed. As might be expected, the ones relating to *slot-definition* metaobjects and *eql-specializer* metaobjects are again the most commonly unsupported.

10 Categorization

The CLOS MOP is sufficiently rich and comprehensive that it is appropriate to categorize and layer the interface it provides to the CLOS language. The AMOP[1] distinguishes two such layers: the introspective layer and the intercessory layer. The first of these, the introspective layer, provides the necessary functionality to retrieve and examine the metaobjects that comprise the elements of a CLOS program. Such a layer is useful on its own to write browser like tools that permit the inspection and analysis of the underlying metaobjects without needing any language customization interface. The second layer, the intercessory layer, is the layer that allows users to modify the behavior of the CLOS language. This includes the necessary interface functions to control the object initialization protocol, the class finalization protocol, the instance structure protocol, the generic function invocation protocol and dependent maintenance protocol as described in *The Common Lisp System Metaobject Protocol*.

The authors of this paper have found it useful to further subdivide the CLOS MOP. Figure 6 shows a more detailed organization of the functionality provided by the Metaobject Protocol. The introspective layer is further divided into two layers: a fundamental introspective layer and an advanced introspective layer.

The first of these is geared towards accessing the more basic properties of the various metaobjects; that is mainly those properties accessible via reader methods for the standard metaobjects. The advanced introspective layer attempts to capture those aspects of metaobjects that relate to other metaobjects as well as those functions which relate to the more sophisticated MOP concepts and protocols. For example, we have chosen to place the reader method *class-name*, which simply returns the name of a class, in the basic introspective layer, but the generic function *class-prototype*, which returns the prototype instance for a class metaobject, has been placed in the advanced introspective layer.

The intercessory layer has been also been divided into a basic intercessory layer and an advanced intercessory layer. The first of these is aimed at capturing those aspects of the MOP that control the creation of objects as well as those elements of the API that are used to attach and detach metaobjects from one another. For example, *make-instance*, *add-method* and *remove-method* are some of the generic functions that can be found in this layer. Finally the advanced intercessory layer serves to capture

Concepts	Basic Introspection	Advanced Introspection	Fundamental Intercession	Advanced Intercession
Classes & Instances	class-default-initargs class-direct-default-initargs class-direct-slots class-direct-subclasses class-direct-superclasses class-name class-precedence-list class-slots	class-prototype class-finalized-p standard-instance-access funcallable-standard-instance-access	(setf class-name) make-instance add-direct-subclass remove-direct-subclass ensure-class-using-class set-funcallable-instance-function	compute-default-initargs compute-class-precedence-list finalize-inheritance validate-superclass allocate-instance
Slot Definitions	slot-definition-allocation slot-definition-initargs slot-definition-initform slot-definition-initfunction slot-definition-name slot-definition-type	slot-definition-readers slot-definition-writers slot-definition-location direct-slot-definition-class effective-slot-definition-class	slot-boundp-using-class slot-makunbound-using-class slot-value-using-class	(setf slot-value-using-class) compute-slots compute-effective-slot-definition
Generic Functions	generic-function-lambda-list generic-function-name generic-function-methods	generic-function-declarations generic-function-method-class generic-function-argument-precedence-order generic-function-method-combination	(setf generic-function-name) ensure-generic-function-using-class	compute-discriminating-function
Methods	method-function method-lambda-list method-specializers method-qualifiers	method-generic-function reader-method-class writer-method-class accessor-method-slot-definition	add-method remove-method make-method-lambda	compute-applicable-methods compute-applicable-methods-using-classes
Specializers		eql-specializer-object specializer-direct-generic-functions specializer-direct-methods	intern-eql-specializer	add-direct-method remove-direct-method
Method combination		find-method-combination		compute-effective-method
Dependents		map-dependents	add-dependent remove-dependent update-dependent	
Lambda lists	extract-lambda-list extract-specializer-names			

Figure 6: Suggested categorization of the CLOS MOP. This decomposes the CLOS MOP along two dimensions. The first dimension consists of four abstraction layers: fundamental introspection, advanced introspection, fundamental intercession and advanced intercession. The second dimension groups components of the CLOS MOP according to which metaobject class and concept they relate to.

the aspect of the interface that is responsible for computing the relationships and the orderings amongst the metaobjects as well as those responsible for storage allocation. For example *compute-class-precedence-list*, which must perform a topological sort of the inheritance hierarchy, and *allocate-instance*, which allocates storage for slots with :instance allocation, have been placed in this layer. Each of these four layers has also been organized according to the various metaobjects manipulated by that layer, thus providing a 2-dimensional perspective of the CLOS Metaobject protocol.

This categorization and decomposition of the CLOS MOP is preliminary and certainly in some cases arguably arbitrary. It represents ideas and work that are in progress and so should by no means be considered definitive.

11 Implementability

A MOP for CLOS walks a fine line between power and implementability. It is obviously desirable to provide a powerful and consistent framework, but it is also extremely important to ensure that the framework can be efficiently implemented without heroic effort from the vendor. Any standardisation effort for a CLOS MOP must take considerations of efficiency very seriously.

Although we have demonstrated that there is a good deal of commonality in the existing implementations, we have not explored the efficiency aspects to the same extent. There may be features in the implementations which, if they are ever used, can cause catastrophic slowdowns to all of CLOS. This needs to be established by experiment and with input from implementors. A good example of the areas which are very critical in this respect are the slot-access protocols, which need to be extremely carefully designed to ensure they can be efficiently implemented.

The Genera implementation of CLOS is interesting in this regard as it can be spectacularly efficient, and this efficiency is not entirely due to the special hardware support that Genera has. Access to slots of CLOS objects, even via accessors rather than *slot-value*, can be almost as fast as an array reference. In order to achieve this performance the implementation has made some tradeoffs in the MOP, and those tradeoffs would be very interesting to examine.

12 Conclusion

In this paper we have briefly reviewed the motivations and structure of the CLOS Metaobject Protocol and examined several popular implementations of CLOS in order to assess which aspects of the MOP are currently implemented and in use. We have provided a proposed decomposition of the functionality captured by the CLOS MOP in preparation of ascertaining the adequacy of this protocol and the possibility for standardization. We conclude that although the CLOS MOP has not yet been standardized by the ANSI X3J13 committee, it is clear that most CLOS implementations provide a fairly comprehensive version of the metaobject protocol. Furthermore, given that few have deviated from the original AMOP[1] specification, at first glance it does not seem unreasonable for the ANSI X3J13 committee to move in the direction of standardiz-

ing the *de facto* CLOS MOP. It should however be noted that the effort presented is preliminary,

References

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

[2] Franz Inc., *Allegro CL User Guide.* Franz Inc., 1996.

[3] Bruno Haible and Michael Stoll. *Implementation Notes for CLISP* Bruno Haible and Mark Stoll, 1999.

[4] Guy L. Steele Jr., *Commmon Lisp: The Language (2nd edition).* Digital Press, 1990.

[5] Richard Corman, *Corman Lisp User Guide and Reference, Version 1.3* Richard Corman, 1999.

[6] The Harlequin Group Limited, *LispWorks Reference Manual, Version 4.0.1.* The Harlequin Group Limited, 1997.

[7] Digitool Inc., *Macintosh Common Lisp Reference.* Digitool Developper Publications, 1996