Specialization Oriented Programming

Jim Newton – Cadence Design Systems

October 4, 2007

1 Abstract

This paper presents an implementation of a generalization of OOP called SOP (Specialization Oriented Programming). Numerous examples are provided of how the system is used both at the meta programming level as well as the application level.

The SOP system presented here is implemented in SKILL, a lisp interpreter product of Cadence Design Systems. The design of the infrastructure is understandable to those familiar with Common Lisp and CLOS (Common Lisp Object System), and have a high-level understanding of the CLOS MOP (metaobject protocol). Although the system's main applications are in Electronic Design Automation (EDA), no understanding of EDA is necessary to understand the concepts presented here.

2 Motivation

The users of Cadence Design Systems custom IC (integrated circuit) tools use the SKILL programming language extensively. Programmers write applications which customize the look and feel of the graphical system, automate the design process by reducing the amount of repetitive work the design engineer must do, and preform time-consuming, tedious verification checks. Other types of programs include automatic layout generation which quickly produce parameterizable layouts which are correct by design. The language has an optional C-style syntax with many engineer-friendly shortcuts which makes it easy for non-programmers to write simple scripts to help in their daily work.

The same language is also a lisp system having the features one would expect such as a REPL (read, evaluate, print loop), a debugger, garbage collection, lexical and dynamic scoping, macros, and lambda functions. As with most lisp systems, the language can be extended though adding functions to the runtime environment.

The SKILL language has a built-in object system called the SKILL++ Object System or simply SKILL++[1]. SKILL++ is conceptually based on CLOS, but fails to provide many of the capabilities of CLOS. Missing are features such as: multiple dispatch, multiple inheritance, method combination, method qualifiers, equivalence specializers, and meta-object protocol. Rather, its features include: single dispatch, single inheritance, `callNextMethod` / `nextMethodP`, class and method redefinition, explicit environment objects, and a per-method choice between lexical dynamic scoping. Also important to note is that while the language itself is interpreted in terms of a proprietary virtual machine, the method dispatch mechanism is implemented in a high performance compiled language. Consequently, generic function calls are as fast as normal function calling protocol.

Because of the limitations of SKILL++ it was desirable to implement a more capable object system which would (1) provide more of the features of CLOS, (2) interface to programs written in the existing SKILL++ system, (3) use OO techniques on existing systems whose object models are out of our control, and (4) be extensible for the types of problems faced in application programming for IC development.

[1]The term SKILL refers to a Cadence proprietary lisp dialect which is integrated into the Cadence IC design software. The term SKILL++ refers to a set of features available in the out-of-the-box SKILL language which provide an object system.

Neither VCAD (an organizational department within Cadence) nor VCAD's customers have access to the SKILL implementation and are prohibited from changing the language itself. Any such extension would need to be provided as a loadable SKILL application. Luckily, because SKILL is a lisp, this can be done in such a way as to seem native to the programmer and invisible to the to the end user.

3 Introduction

A certain progression of generality can be observed in Object Oriented programming system development. This trend can be observed historically in language design, but also is relevant to OO application development to a great extent. First is the message passing approach where objects are able to listen for certain messages, which sometimes contain axillary data, and perform actions consequent to those messages. In this approach problems are solved by defining message protocols which allow objects to make the necessary calculations to arrive at the desired solutions.

Second is the class based approach in which the programmer attempts to isolate and encode common behavior for groups or classes of objects. In these systems it is often the case that programmers attempt to embed the complete behavior of a object group into opaque classes. In such an approach, problems are solved by partitioning capabilities into classes whose objects can manipulate the domain data as necessary.

Third is the generic function approach. The programmer in these systems attempts to **describe the consequence of invoking a generic function**. Problems are solved in this approach by assuring that when a generic function is called, the correct methods get triggered in the correct order.[2]

A fourth generalization of OOP is SOP. Problems are solved in Specialization Oriented Programming by describing method applicability in terms of specializers which are appropriate for the target data. Object systems such as CLOS provide predetermined specialization types such as class based and equivalence based. However, not all data can be described adequately in terms of classes or equivalence.

[2]In CLOS, this is achieved by determining three things: (1) which of the predefined methods are applicable, (2) what is their order from most specific to least specific, and (3) which of the methods are automatically evaluated. There is also some ability to affect control flow by incorporating such tools as `call-next-method`, method qualifiers, and method combinations.

```
(defgeneric Walk (expr env call-stack)
   (:generic-function-class
      sop-cons-generic-function))
```

Figure 1: Code Walker Generic Function

In this development of an SOP system, called VCLOS (VCAD Common Lisp-like Object System), the concept of specializer has been generalized so that class based and object (equivalence) based specializers are two very important special cases. The VCLOS MOP also allows application-specific specialization.

4 Example: Application Programming

The following excerpts are from a code walker, named `Walk` written using SOP style. The code walker examines code written in a particular lisp dialect and reports unbound and unused variables. For purposes of simplicity, the illustrated implementation uses a fictitious Common Lisp-like syntax.

The goal of this illustration is to show that a solution can be more elegant when the language is expressive enough to easily describe wider ranges of data. The goal is not to convince the reader that a particular type of specializer such as the `CONS` specializer itself is a good idea. As with any pedagogical example, the same application could be written many different ways, even without such a extensible specializer approach.

The form (figure 1) declares the generic function to use the `generic-function-class` named `sop-cons-generic-function` which is assumed to already exist. The following sections of this paper discuss one way such a generic function meta-class has been implemented.

The implementation of Walk contains four conceptual parts:

- a recursion engine which includes a termination condition and error handling,
- code to recognized variable references and mark bindings as *used*,
- code to ignore all irrelevant forms encountered during the recursion, and
- code to handle special forms.

The main engine of the code walker (figure 2) starts at a top level expression. If the expression is a list, it calls itself recursively on the elements of the list – with a few notable exceptions. Some of the necessary exceptions can be handled by equivalence specializers such as `(eql t)` and `(eql nil)`. Lisp special forms, such as `(quote ...)` and `(lambda ...)`, cannot be described by equivalence specializers but can be with `CONS` specializers.

Next is the traversal engine based on the class specializer `list` and the termination condition based on an equivalence specializer `(eql nil)`. Thus the engine keeps traversing the lists until they are exhausted. There is also a method specializing on class `t` which will be called if something is encountered which the code walker cannot otherwise handle. The job of the methods that follow will be to assure that everything that occurs in the traversal is handled by an appropriate method and this `"invalid expression"` never gets printed.

When a symbol is encountered the method in figure 3 is applicable. A check is made to see whether the variable is bound in the environment[3]. If so, remember the call stack in the `used` slot of the binding object to keep a record of all the uses of the variable. If the variable is unbound, then print a diagnostic message informing the user where the reference to the unbound variable was made.

[3]The implementation of the `find-binding` function is omitted. It returns a *binding* object by searching for a named variable in a given *environment* object. Such a binding object has a *setf-able* accessor named `used` which can hold a list of call stacks indicating where the code references the variable binding.

```
(defmethod Walk ((expr list)
                 env
                 call-stack)
   (let ((call-stack (cons expr call-stack)))
      (Walk (car expr) env call-stack)
      (Walk (cdr expr) env call-stack)))

(defmethod Walk ((expr (eql nil))
                 env
                 call-stack)
   nil)

(defmethod Walk ((expr t) env call-stack)
   (format t
           "invalid expression ~A: ~A: ~A~%"
           (class-name (class-of expr))
           expr
           call-stack))
```

Figure 2: Recursion Engine and Termination Condition

```
(defmethod Walk ((var symbol)
                 env
                 call-stack)
   (if-let (binding (find-binding env var))
      (setf (used binding) call-stack)
      (format t
              "unbound: ~A: ~A~%"
              var
              call-stack)))
```

Figure 3: Check The Binding of Symbols

```
(defmethod Walk ((expr string)
                 env
                 call-stack)
   nil)

(defmethod Walk ((expr number)
                 env
                 call-stack)
   nil)

(defmethod Walk ((expr (eql t))
                 env
                 call-stack)
   nil)
```

Figure 4: Ignore Certain Atoms

Figure 4 shows how ertain types of self-evaluating atoms encountered such as strings, numbers, and the symbol `t` are simply ignored when searching for variable references.

We now implement some of the special forms. Note that `QUOTE` and `LAMBDA` are not special forms, they are simply symbols which evaluate as any other symbol. If one of these symbols is encountered in a context where it is used as a variable, the code walker with treat it as such. This means we cannot write a method for `Walk` specializing on `(eql QUOTE)`[4] or on `(eql LAMBDA)`. However, lists whose first elements are `QUOTE` or `LAMBDA` are special and must be intercepted before the walker reaches the `QUOTE` and `LAMBDA` symbols themselves.

The `CONS` specializer provides a mechanism for making a method applicable for such a list. Figure 4 implements a method which is applicable if its first argument is a list whose first element is the symbol `QUOTE`. Since an evaluator would simply return the second element of this special form unevaluated, there can be no variable references inside it; so the code walker simply returns `nil`.

Figure 4 implements a method to handle a list whose first element is `LAMBDA`. This method creates new bindings as indicated by the lambda list, and walks the body of the `LAMBDA` with those bindings in place. After the code walker returns from walking the lambda body we can report if any of the new bindings were not referenced by the walked code.[5]

This implementation of the code walker allows us to have a single generic function, `Walk`, whose methods specialize on all of the different types of forms that must be handled differently.

```
(defmethod Walk ((form (cons (eql QUOTE)))
                 env
                 call-stack)
   nil)
```

Figure 5: Handling Special Form (QUOTE ...)

5 Synopsis of Implementation

Being inspired by *Object-Oriented Programming in Common Lisp - A Programmer's Guide to CLOS* by Sonya E. Keene, I decided to implement a CLOS like system for SKILL++. As mentioned before SKILL++ supports single inheritance class definition and single dispatch. In addition the SKILL language provides a powerful `defmacro` facility. These building blocks seemed sufficient to implement the meta-objects necessary to represent the dispatch mechanism in CLOS, most notably: generic function meta-objects, and method meta-objects. SKILL macros were defined to hide some of the implementation details and to make generic function and method definition look very similar to that of CLOS.

A meta-object protocol was also needed. As the basic functionality would be implemented as SKILL++ methods on the standard generic function meta-class and the standard method meta-class, the SKILL++ class inheritance mechanism gave me a Meta-object

[4]Notice that unlike Common Lisp, here the argument of the `EQL` specializer is unevaluated, thus does not need to be quoted. `(eql QUOTE)` is correct, rather than `(eql 'QUOTE)`

[5]The implementations of the functions `derive-bindings-from-ll` and `make-environment` are omitted for this illustration as they do not aid in understanding extensible specializers. The `derive-bindings-from-ll` function returns a list of *binding* objects from a lambda list. The `make-environment` function allocates a new *environment* references the given list of binding objects, and also references the given parent environment.

```
(defmethod Walk ((form (cons (eql LAMBDA))) env call-stack)
   (destructuring-bind (lambda lambda-list &amp;rest body) form
      (let ((bindings (derive-bindings-from-ll lambda-list)))
         (dolist (form body)
            (Walk form
                  (make-environment bindings env)
                  (cons form call-stack) ))
         (dolist (bind bindings)
            (unless (used bind)
               (format t "unused: ~A: ~A~%"
                       var (used call-stack)))))))
```

Figure 6: Handling Special Form (LAMBDA ...)

protocol almost for free. Users of the system (system meta-level programmers) would be able to make SKILL++ subclasses of the standard generic function class or standard method class and specialize alternative behavior in terms of SKILL++ methods, calling the SKILL++ `callNextMethod`[6] as necessary.

The resulting system developed over several years and including incremental phases of development:

1. Generic functions
2. Multiple dispatch
3. Meta-object protocol
4. Method precedence
5. ClosCallNextMethod
6. Method qualifiers: before, after, around
7. Generic specializers
8. Equivalence specializers
9. Application extensible specializers

The Art of the Metaobject Protocol (by Gregor Kiczales and others) was used as a guide–but not adhered to explicitly. In many cases, concessions were made to accommodate the SKILL language, and also some normal functions in AMOP were implemented in VCLOS as generic functions to enhance extensibility. The fundamental conceptual difference between the implementations of the CLOS MOP and VCLOS is that **many concepts which in CLOS MOP are implemented in terms of the *class meta-class*, have been implemented in the VCLOS MOP in terms of the more general *specializer meta-class*.** Notably:

(1) The VCLOS classes `ClosClassSpecializer`[7][8] and `ClosEqvSpecializer` are both subclasses of `ClosSpecializer`.
Users are encouraged to create other subclasses of `ClosSpecializer`.

(2) The CLOS function `compute-applicable-methods-using-classes` has been replaced with `ClosComputeApplicableMethods-UsingSpecializers`.

(3) CLOS memoizes applicable methods with a hash key based on the list of class names of the required arguments. VCLOS memoizes as a function of the return value of the generic function `ClosComputeSpecializerNames` called on the generic function meta-object and the list of required arguments.

[6]The SKILL language reader interprets infix operators. `foo-bar` parses as `(difference foo bar)`. This makes it difficult to imbed special characters such as `-` and `*` into symbols. Consequently it is convention to use camel case names such as `callNextMethod` rather than hyphenated names such as `call-next-method`.

[7]The SKILL language does not have a package system. However, programmers define packages by convention by prefixing global variable names, function names, and class names with a common prefix. In this case the `Clos` prefix denotes functions and classes in the VCLOS SKILL *package*.

[8]SKILL symbols are case sensitive. `EQV` and `eqv` are two different symbols.

`ClosComputeSpecializerNames` returns class names for `ClosClassSpecializer` objects.

As it should be enlightening to the reader, figure ?? shows the the simple implementations of `ClosComputeSpecializerNames` and the corresponding function `ClosComputeSpecializerName`.[9] [10] `ClosStdGenericFunction` only supports class specializers, but `ClosSpecGenericFunction` supports generalized specializers. The function `ClosComputeSpecializerNames` is called when the run-time arguments to a generic function when the generic function is envoked. [11]

The details that follow do not provide low level details of how the VCLOS MOP was designed, but rather how to use the system to implement a new specializer such as the `CONS` specializer used in the code walker illustrated above.

6 Details of Defining a New Specializer: SOP Meta Level Programming

To define a new type of specializer, the programmer must use the VCLOS MOP to define several things:

- How to recognize the syntax of a specializer in a method declaration.
- How to compare (sort in order) this type of specializer to previously existing specializers.
- How to compare two specializers of the same type.
- How to determine whether an object matches the specializer.

[9]The function `ClosGetSpecializerAlist` is an accessor which returns an assoc list mapping the required parameters to the generic function to the set of actual specializers declared for that parameter in all the declared methods of the generic function. This list is maintained as a side effect of the `ClosDefMethod` machinery.

[10]The macro `VcadFirstST` declares a local iteration variable and returns the first element of a given list which makes the given expression (which is normally in terms of of that iteration variable) true.

[11]The Skill idiom `(foreach mapcar (v1 v2 v3) expr1 expr2 expr3 ...)` is equivalent to `(mapcar (lambda (v1 v2 v3) ...) expr1 expr2 expr3)`.

```
(ClosDefMethod foo ((obj (cons string)))
  ...)
```

Figure 8: Example Method Declaration

```
;; less specific than the one above.
(ClosDefMethod foo ((obj list))
   ...)
```

Figure 9: Another Method of foo

First we look at how to define a specializer. Thereafter, well look at defining a new type of generic function which will be able to recognize the specializer.

We would like to declare methods using `CONS` specializers with the SKILL syntax as in figure 6.

The code in figure 6 should have the meaning that if the argument of the generic function `foo` is a list whose first element is a string, then this method is applicable. The method is more specific that a method specializing on the `list` class (figure 6), and less specific than a method specializing via an equivalence specializer on a particular list such as `("hello")`.

Furthermore, wed like the declaration to work intuitively when used recursively as shown in figure 6.[12]

The following method should be applicable if its argument `obj` is a list whose first element is equivalent to 42: e.g., `(42 23 16 15 8 4)`.

The method defined in figure 6 should be applicable if its first argument is a list whose first element is again a list whose first element is equivalent to 42: e.g., `((42 23) (16 15 8) (4))`.

[12]SKILL uses the function `eqv` which is similar to the Common Lisp `EQL` function.

```
(ClosDefMethod foo ((obj (cons (eqv 42))))
  ...)
```

Figure 10: Recursive CONS Specializers

```
(defmethod ClosComputeSpecializerNames ((gf ClosStdGenericFunction) args)
  (foreach mapcar (_param arg) (ClosGetRequiredParams gf)
                               args
          (ClosClassNameOf arg)))

(defmethod ClosComputeSpecializerNames ((gf ClosSpecGenericFunction) args)
  (foreach mapcar (_param arg sublist) (ClosGetRequiredParams gf)
                                       args
                                       (ClosGetSpecializerAlist gf)
          (ClosComputeSpecializerName gf arg (car sublist) (cdr sublist))))

(defmethod ClosComputeSpecializerName ((_gf ClosSpecGenericFunction) arg _param specializers)
  (VifLet (spec (VcadFirstST sp specializers
                             (ClosArgMatchesSpecializerP sp arg)))
    (ClosGetName spec)
    (className arg)))
```

Figure 7: The Implementations of ClosComputeSpecializerNames and ClosComputeSpecializerName

```
(ClosDefMethod foo ((obj (cons (cons number))))
  ...)
```

Figure 11: Recursive CONS Specializers

7 Example Implementation of CONS specializer

The following examples show the steps of using the VCLOS MOP to implement `CONS` specializers in SKILL.

7.1 Class definition

To declare the existence of a new specializer we must make a subclass of `ClosSpecializer` as shown in figure 7.1. The purpose of the slots, `enclosedSpecializer` and `classPrecedenceList` will be explained layer[13][14].

[13]In SKILL, slot information is indicated by `@initarg`, `@initform`, `@reader`, and `@writer`. There is no built-in `setf` mechanism and consequently no `@accessor` mechanism. The VCAD SKILL framework does provide a `Vsetf` macro and the VCAD `ClosDefClass` implements `@accessor`. However, this paper does not explain the development or usage of `ClosDefClass`.

[14]We will use the SKILL package prefix `Sop` to indicate functions, classes, and global variables defined in this application but not native to the VCLOS package.

```
(defclass SopConsSpecializer (ClosSpecializer)
  ((enclosedSpecializer
    @initarg enclosedSpecializer
    @reader SopGetEnclosedSpecializer
    @writer SopSetEnclosedSpecializer)
   (classPrecedenceList
    @initform nil)))
```

Figure 12: The Specializer Meta-Class

7.2 Generic Function Definition

In order to ever hope to use the specializer, we need a generic function class which knows about it. In this case we define `SopConsGenericFunction` (figure 7.2) to be a SKILL++ subclass of `ClosSpecGenericFunction` which is already a subclass of `ClosStdGenericFunction`[15] .

[15]The difference between the `ClosSpecGenericFunction` and the `ClosStdGenericFunction` is that the latter memoizes according to the algorithm in AMOP; i.e., it memoizes based on classes of its arguments. This is very fast because the `classOf` function is an atomic operation in SKILL.

```
(defclass SopConsGenericFunction
          (ClosSpecGenericFunction)
  ())
```

Figure 13: The Generic Function Meta-Class

7.3 The Method ClosAvailableSpecializers Specializing on Generic Function Class

Next, we must describe how the new specializer compares to the other specializers understood by `ClosSpecGenericFunction`. If there are applicable methods containing equivalence specializers, class specializers, and `CONS` specializers, how should they be sorted? This is done by specializing a SKILL++ method `ClosAvailableSpecializers` on the generic function class `SopConsGenericFunction` (figure 7.3).

7.4 Several Methods Specializing on the Specializer Class

Next, we need to tell the function which parses the `ClosDefMethod` that the syntax `(cons ...)` denotes a `SopConsSpecializer`; i.e., a `CONS` specializer; for example `(cons list)` and `(cons (eqv 42))`. We must also provide a mechanism for saving and retrieving the sub-specializer component. As we eluded to above a `CONS` specializer may reference any other specializer name which is itself available to the generic function. In the cases of `(cons list)` and `(cons (eqv 42))` the sub-specializer names are `list` and `(eqv 42)` respectively. The syntax pattern matching is done by the generic function `ClosMatchesSpecializerSyntaxP`. We must specialize a method of `ClosMatchesSpecializerSyntaxP` (figure 7.4) on the SKILL++ specializer class[16] .

The specializer name is destructured with the generic function `ClosSetSpecializerData` (figure 7.4). Notice that `SopGetEnclosedSpecializer` and `SopSetEnclosedSpecializer` are already declared as accessors to the `enclosedSpecializer` slot of the `SopConsSpecializer` class. The `ClosSetSpecializerData` method assures that the slot is initialized.

Next, (figure 7.4) we must tell the discriminating function how to decide whether an argument passed to a generic function invocation matches the specializer[17].

The method `ClosAvailableSpecializers` provides the discriminating function with sufficient clues to sort applicable methods of different specializers. We still need to provide the necessary information for determining which of several like-specializers is more specific or less. The discriminating function calls the function `ClosCmpLikeSpecializers` for this purpose (figure 7.4).

Our implementation here calls the function `ClosCmpSpecializers` which needs an implementation of `ClosGetClassPrecedenceList`. Since this is a computationally expensive calculation the method memoizes the calculated value in the specializer slot `classPrecedenceList`.

The code in figure 7.4 uses several SKILL idioms[18] [19] [20] which might not be immediately understandable to the reader.

For example: the function `ClosGetClassPrecedenceList` calculates the class precedence list (which might be more accurately called specializer precedence list) of `(cons fixnum)`to be `((cons fixnum) (cons number) (cons primitiveObject) (cons systemObject) (cons t) list primitiveObject systemObject t)`.

[16]Given a specializer name such as `(cons (eqv 42))` and the generic function meta-object, the function `ClosNameToSpecializer` efficiently retrieves the specializer object (instance of some subclass of `ClosSpecializer` of the given name.

[17]The SKILL `dtpr` function returns TRUE if its argument is a non-nil list.

[18]`foreach` is a primitive in SKILL. The idiom `(foreach mapcar x some_list exprs ...)` is equivalent to `(mapcar (lambda (x) exprs ...) some_list)`

[19]SKILL keyword symbols are prefixed by `?` rather than `:` as in Common Lisp. `?foo` is a symbol that will always evaluate to itself.

[20]The `VcadAndNotList` function creates a new list containing the elements which are in the first list AND NOT in the second list.

```
;; Define the specializer precedence list for SopConsGenericFunction
;; to be
;;  1. ClosEqvSpecializer         (most specific)
;;  2. SopConsSpecializer
;;  3. ClosClassSpecializer       (least specific)
(defmethod ClosAvailableSpecializers ((gf SopConsGenericFunction))
  '(ClosEqvSpecializer
    SopConsSpecializer
    ClosClassSpecializer))
```

Figure 14: The ClosAvailableSpecializers Method

```
;; E.g., return TRUE if specializer_name is something like (cons number)}
(defmethod ClosMatchesSpecializerSyntaxP ((specializer SopConsSpecializer)
                                          specializer_name)
  (and (listp specializer_name)
       (eq 'cons (car specializer_name))
       (cdr specializer_name)
       (null (cddr specializer_name))
       (ClosNameToSpecializer (ClosGetGenericFunction specializer)
                              (cadr specializer_name))))
```

Figure 15: The ClosMatchesSpecializerSyntaxP Method

```
(defmethod ClosSetSpecializerData ((spec SopConsSpecializer)
                                   specializer_form)
  (SopSetEnclosedSpecializer spec
                             (ClosNameToSpecializer
                                (ClosGetGenericFunction spec)
                                (cadr specializer_form))))
```

Figure 16: The ClosSetSpecializerData Method

```
(defmethod ClosArgMatchesSpecializerP (( spec SopConsSpecializer) arg)
  (and (dtpr arg)
       (ClosArgMatchesSpecializerP (SopGetEnclosedSpecializer spec)
                                   (car arg))))
```

Figure 17: The ClosArgMatchesSpecializerP Method

```
(defmethod ClosCmpLikeSpecializers (( spec1 SopConsSpecializer) spec2 gf param spec)
   ;; required by ClosCmpLikeSpecializers protocol to check for errors
  (callNextMethod)
  (ClosCmpSpecializers gf
                       (SopGetEnclosedSpecializer spec1)
                       (SopGetEnclosedSpecializer spec2)
                       param
                       spec))
```

Figure 18: The ClosCmpLikeSpecializers Method

```
(defmethod ClosGetClassPrecedenceList ((specializer SopConsSpecializer))
  (or (slotValue specializer 'classPrecedenceList)
      (let ((cpl (ClosGetClassPrecedenceList (ClosFindClass 'list))))
        (Vsetf (slotValue specializer 'classPrecedenceList)
               (append (VcadAndNotList
                          (foreach mapcar cl (ClosGetClassPrecedenceList
                                                (SopGetEnclosedSpecializer
                                                     specializer))
                             (makeInstance 'SopConsSpecializer
                                 ?enclosedSpecializer cl))
                          cpl)
                       cpl)))))
```

Figure 19: The ClosGetClassPrecedenceList Method

7.5 The Method ClosGetSpecializer-Comparators Specializing on the Generic Function Class

The discriminating function must look at all the methods defined for a generic function for a given invocation and decide which ones are applicable. It can do this efficiently because it can memoize the results in most cases.

The function `ClosMatchSubSpecializer` calls `ClosGetSpecializerComparators` when trying to figure out whether an object passed to a generic function invocation (called the argument) matches the corresponding parameter of a declared method.

Every method `ClosGetSpecializerComparators` must return a list of a certain format. It is a list of sublists. Each sublist is a list of objects called *comparators*, each an instance of `ClosSpecializerComparatorClass`. These comparators are created by the factory function (actually a macro) `ClosDefComparator`. Each comparator specifies code for making a certain type of applicability decision in order to filter the list of methods into the list of applicable methods.

When the discriminating function is comparing a given generic function argument to a specializer (`meth_spec`) for a method in question it finds the most specific specializer declared for that argument (`arg_spec`). It determines `arg_spec` by using the method `ClosArgMatchesSpecializerP` described above. Once the pair `(meth_spec arg_spec)` has been found a unique comparator can be identified by examining the return value of `ClosGetSpecializerComparators` (and memoizing the result). The discriminating function evaluates the specified code provided by the comparator to determine whether the method is applicable as far as that argument is concerned[21] .

This is a complicated explanation so an example should help. Consider a generic function `foo`, figure 7.5, being called with argument `42`. Of the methods shown, the most specific specializer for the argument is `(eqv 42)`. The discriminating function needs to be

```
(ClosDefMethod foo ((obj (eqv -42)))
  (max 100 (ClosCallNextMethod)))

(ClosDefMethod foo ((obj (eqv 42)))
  (max 100 (ClosCallNextMethod)))

(ClosDefMethod foo ((obj number))
  200)

(ClosDefMethod foo ((obj string))
  300)

(foo 42)
```

Figure 20: Methods With Different Specializers

able to look only at `(eqv 42)` and all from the list of the methods determine which are applicable without actually having access to the argument `42` itself[22].

It must ask the following questions. Is the specializer `number` applicable if `(eqv 42)` is the most specific specializer? YES. Is the specializer `string` applicable? NO. Is the specializer `(eqv -42)` applicable? NO.

The method `ClosGetSpecializerComparators` for the `SopConsGenericFunction` generic function is implemented in figure 7.5. The method simply conses a predefined sublist of `ClosSpecializerComparatorClass` instances onto the list returned from `ClosGetSpecializerComparators` called on the superclass. The tricky thing is to define the list `SopGvConsSpecializerComparators` (figure 7.8). Doing so is a non trivial task.

According to `ClosAvailableSpecializers` above, there are only three specializers available for use on methods. These are `ClosEqvSpecializer`, `SopConsSpecializer`, and `ClosClassSpecializer`. Therefore, `SopGvConsSpecializerComparators` must provide a mechanism for comparing `SopConsSpecializer` to each of these. Thus `SopGvConsSpecializerComparators` is a list of length three. This list is built by calling `ClosDefCom-`

[21] In the case of multi-methods (multiple dispatch), a method is only applicable if it is applicable as far as all required arguments are concerned.

[22] The hash key of the memoizer must not depend explicitly on the arguments; for the potential set of arguments is huge. Rather it should depend on the small set of declared specializers. Otherwise, memoization would not be practical.

```
(defmethod ClosGetSpecializerComparators ((gf SopConsGenericFunction))
  (cons SopGvConsSpecializerComparators
        (callNextMethod)))
```

Figure 21: The ClosGetSpecializerComparators Method

`parator` three times (see NOTES A-1, B-1, and C-1 in the example.) and collecting the results into a list. The order of the elements does not matter.

The purpose of the three comparators are to answer three possible questions respectively. Given the most specific specializer for the argument to the generic function, and a specializer for a method in question, is the method applicable as per that argument? To emphasize, the question must be answered without having access to the argument itself, but rather to the most specific specializer of the argument.

7.6 A: Comparator for SopConsSpecializer vs. SopConsSpecializer

The comparator declaration at A-1 handles the case that the argument to the generic function is known to match a particular `CONS` specializer and asks whether it follows that it also matches another method specific `CONS` specializer.

For example: Suppose the most specific specializer matching an argument is `(cons fixnum)`; meaning that the argument is a list whose first element is a `fixnum`. And suppose we want to know whether a method with a declared specializer `(cons number)` is applicable. The code at A-2 asks whether it is a `list` whose first element is a `number`? The answer is YES.

7.7 B: Comparator for SopConsSpecializer vs. ClosEqvSpecializer

The comparator declaration at B-1 handles the case that the argument to the generic function is known to match a particular equivalence specializer and asks whether it follows that it also matches a method specific `CONS` specializer.

For example: Suppose the most specific specializer matching an argument is the equivalence specializer `(eqv (1))`; meaning the argument is equivalent to `(1)`. And suppose we want to know whether a method whose declared specializer is `(cons number)` is applicable. The code at B-2 asks whether it is also a list whose first element is a `number`? The answer is YES.

7.8 C: Comparator for ClosClassSpecializer vs. SopConsSpecializer

The comparator declaration at C-1 handles the case that the argument to the generic function is known to match a particular `CONS` specializer and and asks whether it follows that it also matches a method specific class specializer.

For example: Suppose the most specific specializer matching an argument is a `CONS` specializer such as `(cons number)`; meaning that the argument is a list whose first element is a `number`. And suppose we want to know whether a method whose declared specializer is `list` is applicable. The code at C-2 asks whether it is also a `list`? The answer is YES.

This concludes the details of the development of the `CONS` specializer.

8 Analysis

In many ways the development shown above, and also in the VCLOS MOP itself is brut force. The implementation is fully functional and extensively tested with Unit tests as well as part of live design projects. Much time has been spent on optimization and refactoring for performance and readability of the code. But of course much more work could be done. It is difficult to create benchmarks which measures performance against similar implementations as I know

```
(defvar SopGvConsSpecializerComparators
  (list
   ;; NOTE A-1
   ;; applicable?    (ClosDefMethod foo ((obj (cons number))) ...)
   ;; most specific: (ClosDefMethod foo ((obj (cons fixnum))) ...)
   ;; application:   (foo (list 1))
   (ClosDefComparator ((meth_spec SopConsSpecializer)
                       (arg_spec SopConsSpecializer))
     ;; NOTE A-2
     (ClosMatchSubSpecializer (ClosGetGenericFunction meth_spec)
                              (SopGetEnclosedSpecializer meth_spec)
                              (SopGetEnclosedSpecializer arg_spec)))

   ;; NOTE B-1
   ;; applicable?    (ClosDefMethod foo ((bar (cons number))) ...)
   ;; most specific: (ClosDefMethod foo ((bar (eqv (1)))) ...)
   ;; application:   (foo (list 1))
   (ClosDefComparator ((meth_spec SopConsSpecializer)
                       (arg_spec ClosEqvSpecializer))
     ;; NOTE B-2
     (and (dtpr (ClosGetData arg_spec))
       (ClosArgMatchesSpecializerP (SopGetEnclosedSpecializer meth_spec)
                                   (car (ClosGetData arg_spec)))))

   ;; NOTE C-1
   ;; applicable?    (ClosDefMethod foo ((obj list)) ...)
   ;; most specific: (ClosDefMethod foo ((obj (cons number))) ...)
   ;; application:   (foo (list 1))
   (ClosDefComparator ((meth_spec ClosClassSpecializer)
                       (arg_spec SopConsSpecializer))
     ;; NOTE C-2
     (memq (ClosGetClassName meth_spec)
          '(list primitiveObject systemObject t)))))
```

Figure 22: Building the Comparators

of no other implementation. But clearly *some* benchmark numbers are needed.

Performance aside, it is clear, that from an API usage perspective, the system does provide a very enlightening abstraction in applications where it is exploited. This results in code that is more expressive and more consistent, and with fewer number of lines needed to implement than would be necessary without the API.

9 Results

The goals of the development of this system were stated earlier.

1. Provide more of the features of CLOS to the Skill programmer.
2. Interface to programs written in the existing Skill++ system.
3. Enable OO techniques on existing systems whose object models are out of our control.
4. Be extensible for the types of problems faced in application programming for IC development.

Goal 1 was achieved by implemented the object system in a CLOS-like style. Using this system, programmers are able to use more advanced features of OO programming and are not limited to single inheritance, single dispatch, and simple method combination.

Goal 2 was achieved by writing the extensions in Skill++ rather than modifying the low level language. Potential users of the system are applications programmers within Cadence. These programmers provide Skill based applications to Cadence's external customers. Any of these users of the Skill language can load the VCLOS library and make use of the enhanced object system. This enhances the business model by allowing application delivery based on higher abstractions that are provided by Skill out-of-the-box.

Goal 3 was achieved by making the specializer capability sufficient to express characteristics of the Cadence data base objects. Because the reader is not assumed to have any understanding of the Cadence data base structure or API, this paper has presented examples of a different nature. However, analogous to the `CONS` specializers described here, the VCLOS system provides a variety of specializers which enable the programmer to describe on the types of non-OO objects which comprise IC design data in the Cadence system.

Goal 4 was achieved by incorporating the concept of specializer in the meta-object protocol of the VCLOS system. When new types of data are encountered, the meta-level programmer can build a generic function class which is able to specialize on characteristics of that data. The method dispatch protocol behaves in an intuitive way when applied to that new data via the domain specific specializers.

10 Recommendations

The enhancements provided to Skill by VCLOS have proved useful, especially the extensible specializers. It seems the type of abstraction made by these specializers would be useful to the CLOS programmer. But it is not clear whether the CLOS MOP is general enough to implement a VCLOS SOP like system. There do indeed seem to be significant obstacles. Nevertheless, these concepts are presented here in hopes that MOP experts consider whether such an implementation would be desirable.

Foreseeable obstacles include the fact that the CLOS system makes some low level assumpations about the behavior of class and equivalence specializers. One of these assumpations is that the `defmethod` macro does not have different implementations for different generic function classes. This makes it difficult for defmethod to recognize different syntax forms for different types of specializers.

Another obstacle is that functions such as `compute-applicable-methods-using-classes` ignore the potential that something more general than classes needs to be considered.

11 Conclusion

There are two dual approaches to abstraction enabling programmers to use a computer language to solve a problem. The first is to transform the domain data into a form which is manipulatable by the computer language (the mountain to Mohammad approach). The second is to transform the language to something that can express truths about the data (the Mohammad to the mountain approach).

Application data does not usually come prepackaged in OO form which the available libraries can manipulate. And systems cannot always be refactored to solve new problems. Rather new programming models must be imposed on old data models. The development shown in this paper is one small way in which lisp-like languages can build abstractions which make problems more easily expressible.

SOP in the form of extensible specializers allow programmers to use object oriented techniques on data that does not fit traditional object oriented views.

12 References

The Art of the Metaobject Protocol; by Gregor Kiczales, et al. 1991 Massachusetts Institute of Technology.

Object-Oriented Programming in Common Lisp - A Programmer's Guide to CLOS; by Sonya E. Keene; Addison-Wesley (1 Jan 1989)

Paradigms of Artificial Intelligence Programming: Case Studies in Common LISP; by Peter Norvig; Morgan Kaufmann Publishers Inc,US (4 Dec 1991)

SKILL: A CAD system extension language; by Tim Barns; Proceedings of the 27th ACM/IEEE conference on Design automation. 1991

SKILL: A Lisp Based Extension Language; by Edwin Petrus. June 4 1993.