The Use of Multimethods and Method Combination in a CLOS Based Window Interface

Hans Muller, hmuller@sun.com
John Rose, jrose@sun.com
James Kempf, jkempf@sun.com
Tayloe Stansbury, tayloe@sun.com

Sun Microsystems, 12-40
Symbolic Computing Department
2550 Garcia Ave.
Mountain View, CA 94043

Topic Area: User Interfaces
Keywords: portable window interface, CLOS, method combination, multimethods

Abstract

Solo is a portable window interface written in the Common Lisp Object System (CLOS) object-oriented programming language. Solo provides a virtual window machine which is targeted to a host window system by implementing a set of host window system specific classes and methods for Solo's host window system driver protocol. The interface presented by Solo to an application insulates it from differences in the host window system, facilitating application portability. Solo distinguishes itself from other object-oriented window systems by exploiting certain features of CLOS. CLOS method combination simplifies initialization of windows while preserving easy extensibility of the basic classes. Generic dispatch on multiple arguments, a feature unique to CLOS, allows a simpler and more flexible input event dispatching protocol. A powerful event description language simplifies the specification of keyboard and mouse events. A prototype implementation runs on the server based X11 and NeWS host systems, and on the frame buffer based Lucid Window Toolkit.

1. Introduction

The increasing demand for easy to use applications has encouraged a proliferation of window systems on a wide variety of hardware and software platforms. There is little agreement, however, on how the programmatic interface to a window system should be designed. From the application developer's viewpoint, this divergence in window interfaces increases the amount of effort needed in porting an application. Many outstanding window-based applications are available only on a single hardware or software platform because window system dependence is designed into the application. Others are available only on platforms for which they were not designed after a long delay .

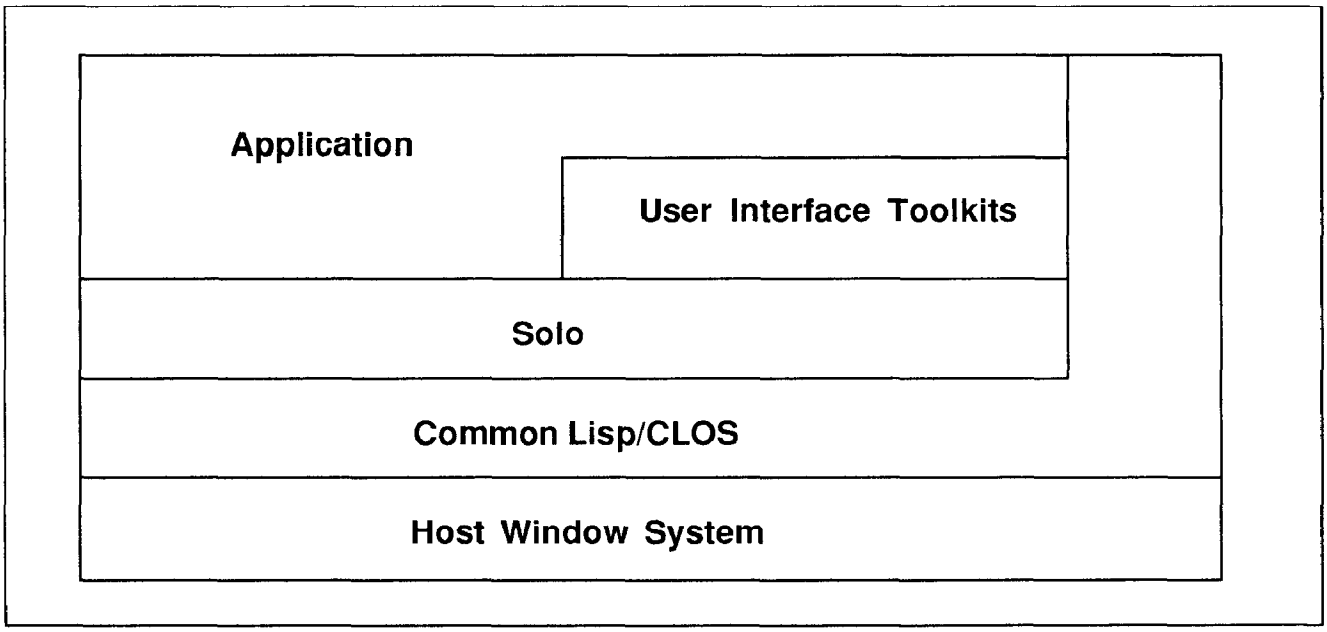
Common Lisp [Steele84] and the Common Lisp Object System (CLOS) [BDGKKM88] are language standards supporting portable object-oriented programming. As such, they provide an excellent substrate for building portable applications. However, the development of window-based applications in Lisp has been hampered by the lack of a suitable programmer's interface. Most portable interfaces, such as Common Windows [Intellicorp86], are not object-oriented; neither are network-oriented interfaces like X [Scheifler86]. Object-oriented interfaces like

that supplied by Symbolics [Symbolics86] are typically weak in the area of portability. Furthermore, many window interfaces dictate aspects of look and feel, or weigh down their windows with complex semantics.

The example of lightweight, server-based window systems, such as X and NeWS [Sun87], and new ideas about how to factor user interface management software into separate toolkit and support layers [Myers89] provided motivation for the design of Solo, an extensible, lightweight portable window interface in CLOS. Solo provides a virtual machine for managing windowing user interfaces across many hardware and software platforms. Fig. 1 illustrates the system architecture into which Solo fits. The Solo virtual machine runs on a host window system, which can be either frame buffer based, such as the Lucid Window Toolkit [Lucid88a] or server based, such as X or NeWS. Complex window semantics (e.g., borders or scrollbars) and other specific look and feel components are implemented by a toolkit layer. Because Solo is implemented in CLOS, it is specifically designed to take advantage of innovative language features provided by CLOS to make both extension and porting straightforward.

In the next section, the basic features of the Solo virtual window machine are presented. Section 3.0 describes how object-oriented programming is used to structure Solo for easy portability. The same mechanisms (inheritance, generic functions) which make customization of Solo for application developers straightforward also facilitate implementation on a new host window system. Section 4.0 describes how Solo uses method combination [Moon86] to facilitate customization of initialization for user defined canvases. In Section 5.0, the event handling system for Solo is described, with special emphasis on how generic dispatching on multiple parameters [Bobrow86] eases customization for user defined canvases. These features distinguish Solo from other object-oriented window systems, since they use language constructs which are unique to CLOS. Section 6.0 presents the event description language for specifying complex input events involving mouse and keyboard state. Section 7.0 makes some comparisons between Solo and other window systems, Section 8.0 briefly

Fig. 1 Solo Window Interface Architecture



describes the current implementation status, and Section 9.0 summarizes the paper.

2. The Solo Virtual Window Machine

The Solo virtual window machine is a collection of generic functions and classes that present an abstract window interface to application programmers. Solo provides a single lightweight window class called a *canvas*. A canvas represents an unadorned rectangular area that serves as a receiver for input events and a destination for graphical output. Canvases can be either transparent or opaque. A transparent canvas does not obscure output to underlying canvases. All canvases are opaque with respect to input events. Canvases have no borders, titles, or other ornamentation, consistent with the design goal of providing the canvas as a simple window component from which more complex components can be built.

Fig. 2 Canvas Class and Protocol

Class Definition:

```
(defclass canvas (drawable event-dispatch display-specific-mixin)
  ((parent :initarg :parent :reader parent)
   (children :initarg :children)
   (depth :initarg :depth :reader depth)
   (bounding-region :initarg :bounding-region)
   (transparent :initarg :transparent :reader transparent)
   (mapped :initarg :mapped :reader mapped)
   (retained :initarg :retained :reader retained)
   (interests :type list :initarg :interests))
  (:default-initargs :parent *default-display* :children nil
                     :transparent nil :mapped nil :retained nil
                     :interests nil :event-dispatch-process nil
                     :event-dispatch-queue nil))
```

Generic Function Protocol:

```
initialize-instance :around canvas &rest args
parent canvas
(setf parent) parent-canvas canvas
children canvas
(setf children) child-canvases canvas
bounding-region canvas
(setf bounding-region) bounding-region canvas
status canvas
(setf status) status canvas
retained canvas
(setf retained) canvas symbol
interests canvas
(setf interests) interests canvas
mapped canvas
(setf mapped) mapped canvas
receive-event canvas interest event
deliver-event canvas interest event
```

Solo maintains canvases in a parent/child hierarchy, using a list of children stored in the parent canvas structure, and a parent pointer stored in the child canvas structure. Children lists are sorted according to occlusion. Child canvases are clipped to the parent's boundary. Each canvas records its parent upon creation. A newly created canvas is placed at the front of the parent's list of siblings. At the root of the parent/child hierarchy is an instance of a host window system specific display class, corresponding to a physical display device like a workstation console. Solo maintains a default display on which canvases are created if no other display is indicated, and multiple displays can be active in Solo at one time. Solo even allows different host window systems to coexist, so an application can access different window systems from within the same Lisp image.

Each canvas has a set of slots which determine its size, placement, parent, children, transparencies, sensitivity to input, and other properties. Fig. 2 contains the definition of the `canvas` class and the method protocol for slot access and event handling. All canvas operations other than input and output are effected by changing the value of some canvas slot. For example, the `bounding-region` slot contains a `region` object which models the physical boundaries of the canvas on the display surface. In order to resize a canvas, a new region with new boundaries is deposited in the `bounding-region` slot. Slot accessor methods on canvas slots like `bounding-region` additionally update other relevant data structures and cause changes in canvas state to be visually manifested, besides simply depositing a new value in the slot or returning the existing value.

By connecting slot access to the update of other data structures (a technique sometimes known as *active values* [Stefik86] [Schaffert86]), methods to expose, bury, move, and resize canvases can be written in terms of updates to canvas slots. The focus on updatable slots in the design makes performing operations on canvases more declarative, since a change to a slot is effected simply by specifying its new value, rather than by giving an algorithm for achieving that value. This is in contrast to the imperative design of other window systems, where the emphasis is on operations (e.g., move, resize) which modify window state, possibly in a complex manner. In imperative designs, user access to window state is often supplied as an afterthought, or sometimes even omitted. As a convenience, Solo provides a library of conventional window operations such as `expose`, `bury`, `destroy`, etc.

In addition, there is an event dispatching protocol associated with the `canvas` class. This event dispatching system delivers mouse events, keyboard events, and other window system generated events to canvases. For example, when an attribute of a canvas (such as its stacking order) is changed asynchronously (i.e., not by application program control), a notification event is generated, containing the name of the slot to be changed and its new value. When this event is delivered to the canvas, the slot value is actually changed. Section 5 contains more detail on the Solo event dispatching system.

The Solo `image` class is a generalization of what some window systems call a bitmap. Images are two dimensional arrays of integer pixels. To make an image visible, it must be copied to a canvas with `copy-area`. Images support two capabilities that canvases do not:

- It is possible to read back the value of a pixel at some location in an image,
- Images can be stored and restored in specially formatted files.

Applications that use server window systems like X or NeWS need to be able to control on which side of the client/server connection image instances are cached, for performance reasons. An application that reads or writes

individual pixels extensively should keep the image on the client side but an application that repeatedly copies an image to a canvas should cache the image on the server side. Applications can advise Solo about where to store image data by setting the `image` object's `host` slot. The value of this slot can be either `:client` or `:server`. Solo may transfer the image between the server and client when the value of this slot is changed.

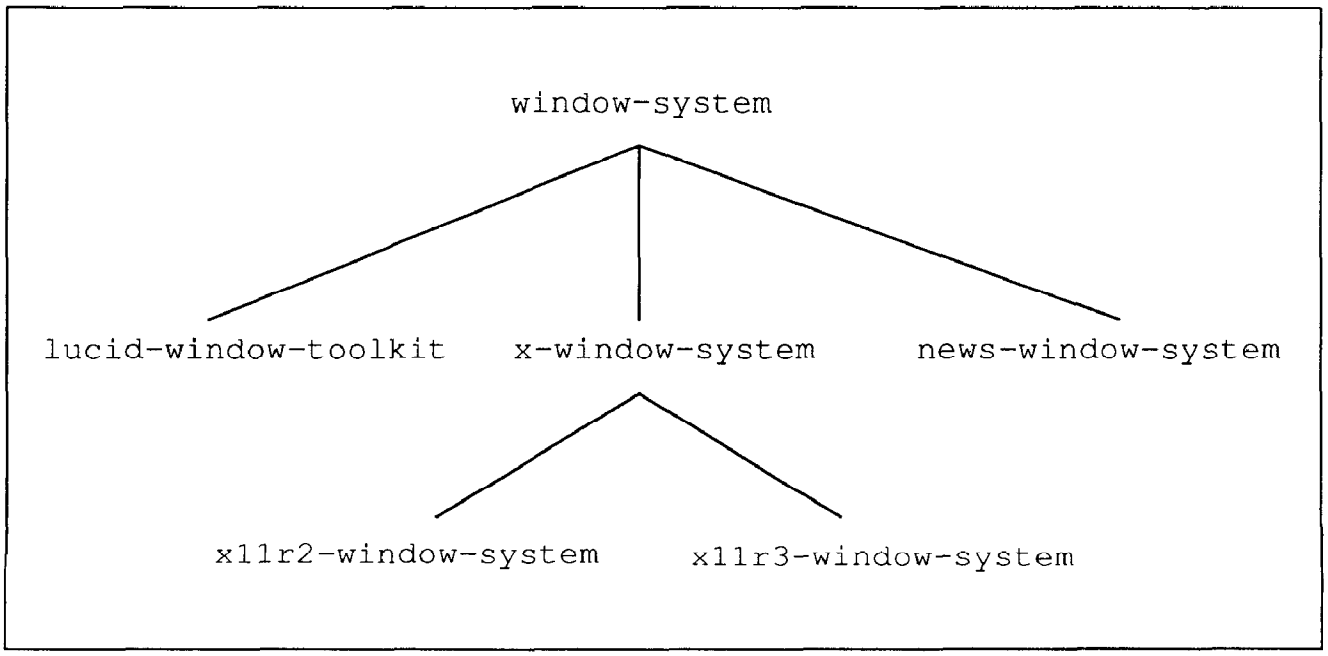
Solo's handling of colors and graphics state, such as line styles, hatching, etc., is entirely conventional and follows that of X11. There are classes modeling colors and graphics context. Method protocols for manipulating graphic attributes and for communicating client wishes to the host window system are available. Fonts are currently handled similarly to X11 Release 3 [Scheifler88b]. While the inclusion of X-specific window features in Solo may seem to bias Solo towards an X host, the protocol presented by Solo to the application is entirely free of any assumptions about the host window system. Whether or not a particular host window system provides semantically equivalent operations will influence the amount of code necessary to port Solo to the host, but applications built on Solo need not be modified.

3. Object-Oriented Host Window System Interface

The emphasis in most other work on object-oriented window systems has been on employing the extensibility provided by inheritance and generic operations to support easier customization of the user interface for applications software. Equally as important, however, is the ease with which the window system software itself can be customized to match different configurations of host graphics hardware and software. With the proliferation of graphics hardware and host window system software, maintaining a window system designed to run on a variety of hardware and software configurations has become analogous to the problem of maintaining a compiler for a variety of machine architectures [Brooks86].

Solo uses inheritance and generic operations to facilitate modularization of host specific code, and even to allow different releases of the same host which have incompatibilities to be

Fig. 3 Host Window Specific Display Classes



accommodated in the same Lisp image. Fig. 3 illustrates the host window system class structure currently available in Solo. A root abstract superclass, `window-system`, provides a common ancestor for all window systems.

Under the root class are subclasses representing the various host window systems. Each host may also have subclasses, representing different releases of the same window system. An example is the `x11r2-window-system` and `x11r3-window-system` classes in the figure. Between X11 Release 2 [Scheifler88a] and X11 Release 3, an incompatibility in the way fonts are specified arose. The incompatibility was accommodated by subclassing the `x-window-system` class with the two subclasses, one for each release, and implementing the host-specific font method protocol differently for each subclass. The window system class instances themselves have little state, just a string identifying their host, and are primarily used to dispatch to the appropriate device driver methods.

Solo specifies a porting interface called the *host window system driver protocol* which must be supported by each window system class. The driver protocol matches the Solo abstract window interface with the underlying host window system. Window system features provided by the host window system need not be reimplemented in Solo. When porting to a new host window system, host-specific methods must be provided for the following operations:

- Global event dispatching,
- Image to host window system transfer,
- Canvas to host window system window,
- Display to host window system display,
- Font handling,
- Color handling,
- Two dimensional graphics.

Naturally, the amount of support a particular host gives for a particular capability will determine the amount of code needed to implement that capability in Solo. In the worst case, if no host window system is available, Solo could be implemented directly on the graphics hardware.

4. Method Combination and Object Initialization

Solo uses method combination internally to simplify application level extensions of class instances having system resources allocated on a server. Initialization of instances is arranged so that window system specific object state is synchronized with the server after all client side initializations, including initializations for subclasses, are finished. Initialization proceeds in two phases. The first phase validates the initial values of slots and performs client-side initializations. The second phase communicates information to the server after other initialization methods, including subclass methods, have completed. In frame buffer based window system, the second phase might be used to flush internally buffered initializations to the frame buffer. Subclasses have the opportunity to customize both before and after server initialization. CLOS method combination provides the linguistic support.

Fig. 4 illustrates an example of how client side objects with server side resources are initialized. The figure diagrams the `initialize-instance` method execution sequence for the `vertical-scrollbar` subclass of `canvas`. In CLOS, the class `standard-object` is the superclass of all standard instances, and it supplies a default `initialize-instance` method. Subclasses typically customize initialization by writing `:after` methods on `initialize-instance`. The `canvas` class itself implements two initialization methods:

- A `:after` method, which handles custom client side initialization for `canvas` alone,
- A `:around` method, which handles communication with the server.

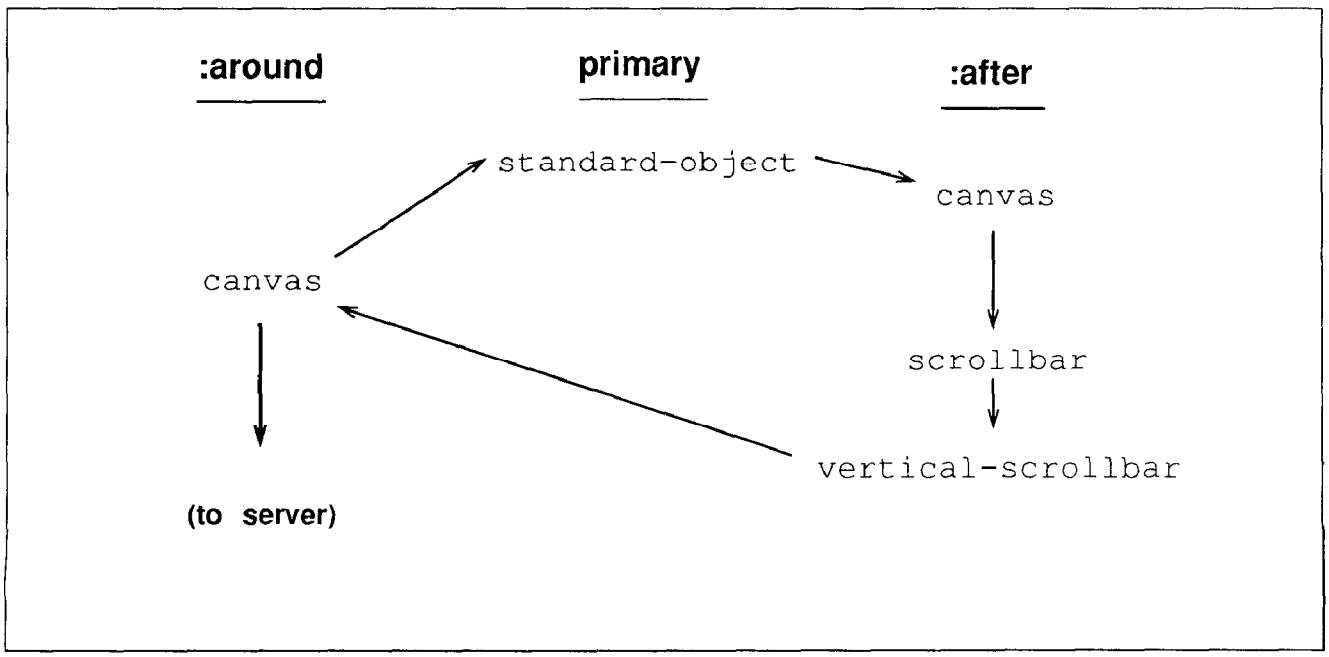
The `:around` method initially does any host window specific actions to start buffering or to tell the server that a new window is coming, then invokes the CLOS `call-next-method` local function. `call-next-method` calls the next most specific method for the generic function invocation, which, in this case, causes the primary/`:after` method complex to be invoked. Within this complex, application specific customizations of `canvas`, such as the `scrollbar` and `vertical-scrollbar` classes in the figure, have customized initialization using the recommended CLOS methodology, namely by writing `:after` methods on `initialize-instance`. These customizations run before server initialization.

After the primary and `:after` methods have run, control returns to the `:around` method, which then finishes initialization of server resources. When server resource initialization is completed, the `:around` method sets the `status` slot in the new canvas to the value `:realized`. If the `canvas` subclass has initializations which must be executed immediately after the allocation of server resources, it defines an `:after` method on the `(setf status)` operation specialized to match `:realized`. As an example of how CLOS `eql` specializers and multiple dispatch simplify customization in Solo, the following code might be part of the `scrollbar` protocol:

```
(defmethod (setf status)
 ((n (eql :realized)) (s scrollbar))
    (call-next-method)
    (post-server-initialization s)
    n)
```

The two-phase initialization process requires that all window classes run their first phase initializations, and then all window classes run their second phases. This control structure can be implemented using CLOS method combination, but not with the simpler "message to super" semantics supported by other object-oriented languages. In addition, subclasses which have special needs for server communication, perhaps as a result of a

Fig. 4 Initialization of a `vertical-scrollbar` *Instance*



particular feature provided by a host window system port, can achieve finer control by customizing `:around` methods on `initialize-instance`.

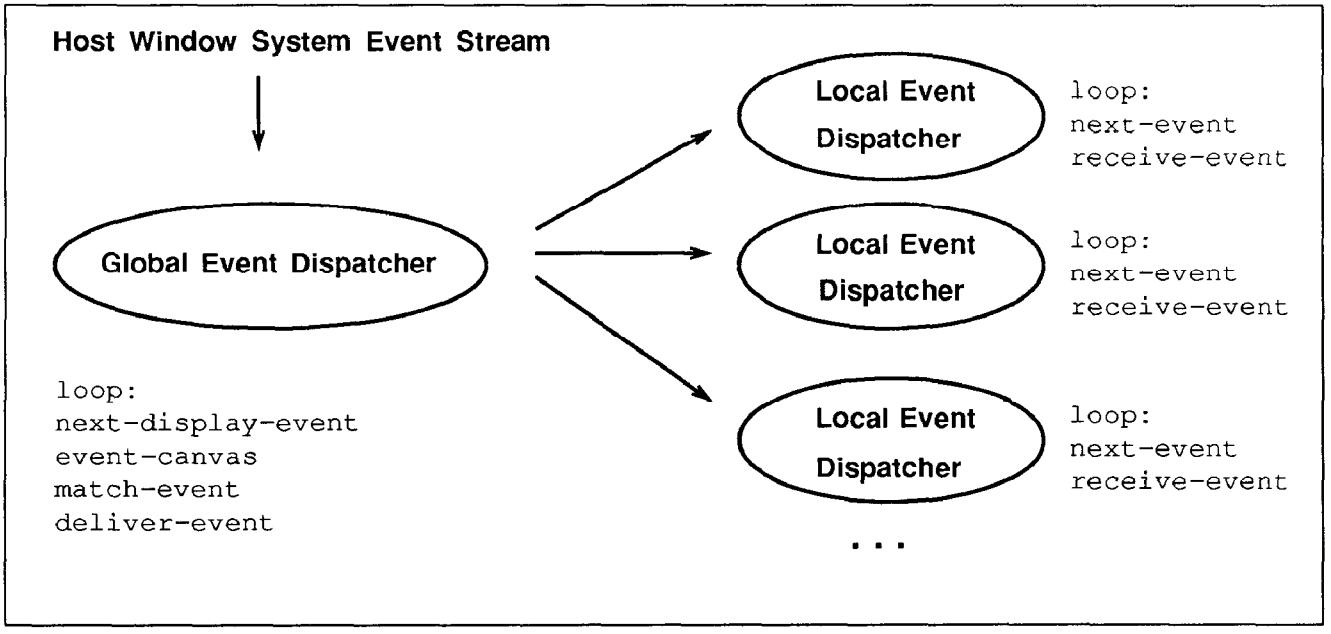
5. Event Handling System

Because Solo mediates between the host window system and the application, it must map user generated events, like changes in mouse and keyboard state, and window system events, like requests to repaint damaged portions of the screen, into application specific code. Other Lisp-based window systems have handled this with special purpose mechanisms like a table of callback functions. Solo uses a generic function dispatching on multiple parameters to implement the mapping. Using the generic function feature of CLOS instead of a special purpose mechanism gains a uniform and commonly understood dispatching design that can trivially be extended to accommodate new event types and incremental changes to event driven application code. In addition, a way to control the flow of events from the window system to the application must be provided, since not all windows will be interested in all the events which they could potentially receive. This is especially critical when the underlying window system is a server, like X or NeWS, because the cost of sending an event between the client and the server is high. Solo uses interest objects to restrict which events are delivered to a particular canvas and to subdivide the event space along application specific boundaries.

Fig. 5 illustrates the overall structure of the event dispatching system in Solo. The event dispatching system is divided into two parts:

- A global event dispatching loop, which receives raw window system events from the host window system, converts them into low-overhead event structures within Lisp, and delivers them to canvases which have expressed interest in them,
- A local event dispatching loop, which queues the low-overhead event structures from the global event dispatcher and runs canvas specific methods in the order events were received.

Fig. 5 Solo Event Handling System



Each display object in Solo has associated with it a global event dispatching loop. The local event dispatching loops may be constructed on a per canvas basis, or a group of canvases may be managed by the same local loop. Although this input architecture is most conveniently implemented with the event dispatching loops in separate lightweight processes, the design does not require multiprocessing support, since the dispatching system could run at interrupt level.

The following code sketches the implementation of the global event dispatcher:

```
(loop
  (let*
    ((e (next-display-event display))
     (c (event-canvas e))
     (i
       (match-event e
                    (interests c))))
    (when i
     (setf (event-interest e) i)
     (deliver-event
       c i e))))
```

Each raw event generated by the underlying window system is received from the display by the generic function `next-display-event` and converted into an event structure. Event objects themselves are small, unnested structures that can be quickly created and quickly reclaimed by ephemeral garbage collection. The `next-display-event` generic function maps the window associated with the event by the host window system into a canvas. The event is matched against the canvas' interest list by the `match-event` generic function. If there is a matching interest, the event is passed along to a local event dispatcher by the `deliver-event` generic function. The specification of the default `deliver-event` method requires it to queue the event for the canvas' local event dispatcher.

Local event dispatching is handled by a loop similar to the following:

```
(loop
  (let ((e (next-event)))
    (receive-event
       (event-canvas e)
       (event-interest e)
       e)))
```

The `next-event` function returns events in the time order they occurred. The event is delivered to the canvas along with the matching interest by the `receive-event` generic-function. The default `receive-event` method dispatches on both the canvas and the interest arguments.

Subclasses of the `interest` class are usually created by specifying a value for the interest's `event-specfication` slot. For example a scrollbar canvas that interpreted a left mouse button down action as "scroll the top of the canvas to here" could define an interest like:

```
(defclass scroll-top-to-here (interest)
  ()
  (:default-initargs
    :event-specification
    '(:mouse (:button :left :down) t)))
```

Placing an instance of `scroll-top-to-here` on the scrollbar canvas' interest list would cause left mouse button down events to be delivered to the application's local event dispatcher. The event specifications are encoded using the event specification language described in the next section.

Customizations of dispatching are possible on both the global and local dispatching levels. On the global level, applications may specialize `deliver-event` to use a different queueing mechanism from the one provided by Solo. For example an application that wanted to handle ^C keyboard interrupts by immediately interrupting the application rather than by queuing the event would create an interest that matched ^C and then define a method on `deliver-event`:

```
(defclass control-c (interest)
  ()
  (:default-initargs
    :event-specification
    '(:keyboard #\control-c)))

(defmethod deliver-event
  (c (i control-c) e)
    (declare (ignore e))
      (interrupt-application c))
```

On the local dispatching level, applications can write methods on `receive-event` to specialize on any combination of the class, interest, and event arguments. For example, to complete the implementation of the scrollbar canvas, the `scroll-top-to-here` interest could be used to specialize `receive-event`:

```
(defmethod receive-event
  (c (i scroll-top-to-here) e)
  "Scroll the top of the canavs to
   (event-y event) in extent
   coordinates"

    ...

)
```

Because CLOS allows specialization of any combination of method arguments, applications which define their own event types can choose to recognize them for application specific canvases by specializing on the new event type:

```
(defmethod receive-event
  ((c spreadsheet) i (e update))

    ...

)
```

or, with only one specializer, thus extending the event repertoire for all canvases:

```
(defmethod receive-event
  (c i (e update))

    ...

)
```

This kind of flexibility is much harder achieve in single dispatch languages, where the choice of which parameter to dispatch on (canvas or event) requires either an explicit coding of the second parameter's type in a `typecase` statement, or that an additional method be defined to relay the second dispatch [Ingalls86].

Finally, applications can send internal events without going through the dispatching loop by calling the nongeneric function `send-event`. This function is a client-callable interface to `deliver-event` which matches the event against the canvas' interest list and, if there is a matching interest, delivers the event to the canvas. In effect, `send-event` and `deliver-event` are two entry points for the same behavior, but they have completely different roles in application code. The role of `send-event` is to be called by clients, to process its arguments slightly, and then to call `deliver-event`, while `deliver-event` is not called directly by clients, but is specialized by client classes. The parameter set of `deliver-event` is designed for convenient specialization rather than for convenient calling.

6. Event Description Language

Event interests for mouse and keyboard events are specified using a sophisticated event description language. The space of window system generated events can be quite large, consisting of various combinations of mouse and keyboard events. For example, the combination of three mouse buttons, two possible actions for each button (up or down), five keyboard modifier keys (control, shift, meta, hyper, and super), and one or two mouse buttons acting as modifiers yields over a thousand events. Most applications will typically only need access to a fraction of these; nevertheless, a concise syntax for specifying input events simplifies the task of customizing event interests.

Event `interest` subclass instances contain a slot called `event-specification` that contains a representation of the set of events matched by the interest. The syntax for specifying event interests is complex, but most of the complexity exists to support precise specification of mouse "gestures", i.e. combinations of mouse movements, mouse buttons, and keyboard modifier keys. The BNF in Fig. 6 specifies the syntax for an event description.

An example from the event specification language syntax is the grammar production *mouse*, for a mouse event. The *action* element subdivides the space of mouse events into three general categories:

- button - a mouse button has gone up or down or a mouse button "click" has occurred. A mouse click occurs when a mouse button goes up and down in about the same spot over a short time interval,
- crossing - the mouse cursor has crossed a canvas boundary,
- move - the mouse cursor has moved.

The *modifier* element specifies the state of the keyboard modifier keys and the other mouse buttons. If *modifier* is `nil` then the event specification will only match events where none of the modifiers are down, if it is `t` then the specification will match mouse events with any combination of modifiers down. If the modifier is a single keyword then the corresponding modifier key must be down. Finally the modifier state may be represented by a logical expression written in terms of `and`, `or`, and modifier keywords. For example, to specify either control-shift or control-meta:

```
(or (and :control :shift)
    (and :control :meta))
```

Note that both the keyboard modifier keys (control, shift, meta, hyper, and super) and the mouse buttons may be used as *modifier* elements. A mouse button may only be used for *modifier* if it is not part of the action.

If the *action* element corresponds to a button transition, the syntax for specifying the transition is:

```
(:button { button-name | t }
         { button-action | t })
```

Specifying `t` for the second or third argument means any button or any button action (up, down, or click). The second argument may also be a keyword that identifies an individual button or an expression that specifies a set of

Fig. 6 Event Specification Language Syntax

```
event := mouse | keyboard | damage | notification | (or {event}+)
keyboard := (:keyboard {character | t} [:up] [:down])
damage := (:damage)
notification := (:notification operations+)
mouse := (:mouse action {modifier | t | nil})
action := button | (:crossing [:enter] [:exit]) | (:move)
button := (:button {button-name | t} {button-action | t})
button-name := :left | :middle | :right |
               (or button-name*) | (and button-name*)
button-action :=  click := :click1 | :click2 :click3 | :click4 |
                           :click | :up | :down | (or button-action+)
modifier := :control | :shift | :meta | :hyper |
            :super | :left | :middle | :right |
            (or modifier*) | (and modifier*)
operations := :status | :mapped | :bounding-region | :children | :parent
```

alternative individual buttons, button chords, or a combination of both. For example to specify either the left or right button moving up or down:

```
(:button (or :left :right)
         (or :up :down))
```

7. Comparison with Other Work

Currently, Common Windows [Intellicorp86] is the most widely used portable window toolkit in Common Lisp. Common Windows has provided some applications with platform independence, but because Common Windows is not object-oriented, it lacks extensibility. This makes user defined extensions of windows hard to do. Inclusion of user interface policy in the Common Windows design also hinders customization. The design is based on the Interlisp-D window toolkit [Xerox85], and windows in Common Windows come bundled with various components, the look and feel of which cannot be changed.

Solo differs from other CLOS-based window systems in a number of ways. Unlike CLUE [Kimbrough88], Solo is not tied to a single host window system, but is designed to be portable. The Solo model of event dispatching requires specialization of at most two (and usually only one) generic functions, rather than many, as in DELI [Pettingill88]. In addition, Common Lisp streams are implemented on top of Solo, not as part of it, while DELI includes a mixin class supporting stream operations as part of the basic window class definition. The goals of Solo are similar to the Window System Independent Interface (WSII) portion of DELI. Solo differs from Silica [Rao88] in that Silica exposes host window system specific components in the application interface. Silica also has a complex semantic model, involving contracts and window trees, in comparison with Solo's rather simple model, and Silica does not specify an object-oriented interface to the host window system as does Solo. CORAL [Szekely88], a CLOS based user interface toolkit, makes heavy use of active values like Solo, but at the toolkit rather than at the window system level.

In relation to object-oriented window systems in other languages, Solo does not provide a complete application framework. The interface implemented by Solo is lower level than the Smalltalk model/view/controller [Goldberg83] or MacApp [Schmucker86], though Solo's event dispatching system could easily support application frameworks of that nature. In general, the Solo design has stressed a simple, lightweight window model and event based input dispatching, to minimize the amount of code at the interface between the application and the host window system.

Solo synthesizes a number of useful ideas from other window systems. The canvas class is similar to that provided by NeWS, as is the basic input model, but the utilization of CLOS allows window system clients to customize window classes even for host window systems running on frame buffers or with nonprogrammable servers, such as X. As mentioned in Section 2, font handling and graphic contexts are similar to X. The window operations are like those of Common Windows.

8. Current Implementation Status

A prototype implementation of Solo has been completed on the X11 server based window systems, and on the Lucid Window Toolkit, a frame buffer based window system which runs on SunView 1 [Sun88a]. The X11 port runs on both X11 Release 2 and X11 Release 3. The prototype was built in Sun Common Lisp 3.0 [Lucid88b]. Sun Common Lisp 3.0 provides a lightweight process mechanism similar to stack groups on the Lisp Machine [Symbolics86], and this mechanism has been used in Solo. The public domain PCL implementation of CLOS [Kiczales88] was used to develop Solo.

In order to test the Solo application interface, the Generic Windows user interface toolkit [Schoen88] was implemented on Solo. Generic Windows provides facilities for user interface programming such as menus, titled windows, scrollbars, etc., similar to Common Windows, which are a level above the simple canvases provided by Solo. No major changes were required in the Solo design, and no features of Generic Windows were left out of the implementation. Finally, the Hyperclass AI application framework [Smith88] was brought up on top of the Generic Windows implementation. Hyperclass provides support for programming complex displays in a frame-based AI language. No significant difficulties were encountered.

A draft specification for the window interface part of Solo has been completed, for Solo users and implementors who would like to port Solo to other hosts. A set of driver protocol methods for the C based X11 library (Xlib) is currently being written. The prototype X11 driver was written for the Lisp based X11 library (CLX) but was found to be too large for supporting serious applications development. In addition, a layer is being added to Solo to support user interface toolkits. Although this layer is designed in a look and feel independent fashion, a binding to the XView implementation of the OPEN LOOK [Sun88b] user interface toolkit is underway. Toolkit components from the XView OPEN LOOK toolkit are being integrated with Solo. A draft specification of the toolkit support layer is also in progress.

9. Summary

The Solo window interface implements a virtual window machine which insulates application programmers developing CLOS applications from the underlying host window system, providing application programmers with portability between host window systems and between releases of the same host window system. The Solo design stresses a simple window abstraction, called a canvas, on top of which applications or toolkit developers can build more complex components. Fonts, colors, and graphics are handled conventionally, in a manner similar to X11, but the application interface remains the same regardless of the host window system.

The input system is sectioned into two parts, a global event dispatcher which handles processing events from the host and a per canvas event dispatcher that canvas subclass developers typically subclass. Active values on canvas slots cause slot update to update dependent state as well. Canvases express interest in particular events which they want delivered, or revoke interest in those events which are no longer of concern. An event description language allows different host window system implementations or application level software to precisely specify the mapping between mouse and keyboard states and generated input events. Event dispatching need not be confined to input, however, and customizations in the direction of event driven interapplication communication are also possible.

Object-oriented design was important in constructing the interface to the host window system. A port of Solo needs to implement a host window system specific class and methods for the host specific device driver generic functions. The driver protocols match the Solo abstract interface to the host window system's event handling, window and display protocol, and graphics capabilities, such as fonts and color. The host driver interface is even flexible enough to handle incompatibilities between releases of the same host window system. Applications on Solo are thus insulated from changes in the host between releases, and, in the case of server based window systems, can communicate with servers running different releases of the host from the same Lisp image. Communication between an application and different hosts from

within the same Lisp image, for example a server based and a frame buffer based system, is also possible.

A prototype implementation of Solo has been developed on X and the Lucid Window Toolkit, using Sun Common Lisp 3.0 and the public domain PCL CLOS implementation. The Generic Windows user interface toolkit, and the Hyperclass AI application framework have been moved to a Solo base. Currently, a set of driver methods for the Xlib X11 library are being written, and a look and feel independent toolkit interface is under development. The toolkit interface is initially being bound to the XView implementation of the OPEN LOOK look and feel. A specification for the window interface layer is available, and one for the toolkit interface layer is in progress.

10. Acknowledgments

The authors would like to thank Steve Gadol for his support of the Solo experimental prototype and Robert Mori, who was instrumental in helping with X11 releases.

11. References

[BDGKKM88] Bobrow, D., DeMichiel, L., Gabriel, R., Keene, S., Kiczales, G., and Moon, D., "Common Lisp Object System Specification," **SIGPLAN Notices, 23**, September, 1988.

[Bobrow86] Bobrow, D., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F., "CommonLoops: Merging Lisp and Object-Oriented Programming," *Proceedings of the First Annual OOPSLA*, **SIGPLAN Notices, 21(11)**, pp. 17-29, 1986.

[Brooks86] Brooks, R., Posner, D., McDonald, J., White, J.L., Benson, E., and Gabriel, R., "Design of an Optimizing, Dynamically Retargetable Compiler for Common Lisp," *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pp. 67-85, 1986.

[Goldberg83] Goldberg, A., and Robsen, D., *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983.

[Intellicorp86] *Common Windows Manual*, Intellicorp, Mountain View, CA, 1986.

[Kiczales88] "The Implementation of PCL," talk given at the First CLOS Users and Implementors Workshop, Oct. 3 & 4, 1988.

[Kimbrough88] *Common Lisp User Interface Environment*, Texas Instruments, Dallas, TX, 1988.

[Lucid88a] *Sun Common Lisp 3.0: The Lucid Window Toolkit*, Lucid Inc. and Sun Microsystems, 1988.

[Lucid88b] *Sun Common Lisp 3.0: Advanced User's Guide*, Lucid Inc. and Sun Microsystems, 1988

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

[Myers89] Myers, B., "User-Interface Tools: Introduction and Survey," **IEEE Software**, pp. 15-23, January, 1989.

[Pettingill88] Pettingill, R., "The Deli Window System: A Portable, CLOS Based Network Window System Interface," *Proceedings of the First CLOS Users and Implementors Workshop*, pp. 121-124, Oct. 3 &4, 1988.

[Rao88] Rao, R., "Silica: A Window System Kernal," *Proceedings of the First CLOS Users and Implementors Workshop*, pp. 125-128, Oct. 3 & 4, 1988.

[Schaffert86] Schaffert, C., Cooper, T., Bullis, B., Kilian, M, and Wilpolt, C., "An Introduction to Trellis/Owl," *Proceedings of the First Annual OOPSLA*, **SIGPLAN Notices, 21(11)**, pp. 9-16, 1986.

[Scheifler86] Scheifler, R., and Gettys, J., "The X Window System," **ACM Transactions on Graphics, 5(2)**, 1986.

[Scheifler88a] Scheifler, R., *X Window System Protocol, Version 11, Release 2*, MIT, Cambridge, MA, 1988.

[Scheifler88b] Scheifler, R., *X Window System Protocol, Version 11, Release 3*, MIT, Cambridge, MA, 1988.

[Schoen88] Schoen, E., Smith, R., and Atkinson, A., *The Generic Window System*, Schlumberger Palo Alto Research, Palo Alto, CA, 1988.

[Schmucker86] Schmucker, K., *Object-Oriented Programming for the Macintosh*, Hayden Books, Hasbrouck Heights, NJ, 609 pp., 1986

[Smith88] Smith, R., Schoen, E., and Atkinson, A., *Metaclass User's Guide*, Schlumberger Technologies, Inc., 1988.

[Steele84] Steele, G., *Common Lisp: The Language*, Digital Press, Marlborough, MA, 1984.

[Stefik86] Stefik, M, and Bobrow, D., "Object-Oriented Programming: Themes and Variations," **AI Magazine**, **6(4)**, pp. 40-62, 1986.

[Sun87] *NeWS 1.1 Manual*, Sun Microsystems, Mountain View, CA, 1987.

[Sun88a] *SunView1 Programmers Guide*, Sun Microsystems, Mountain View, CA, 1988.

[Sun88b] *Open Look Graphical User Interface Functional Specification*,, Sun Microsystems, Mountain View, CA, 1989.

[Szekely88] Szekely, P, and Myers, B., "A User Interface Toolkit Based on Graphical Objects and Constraints," *Proceedings of the Third Annual OOPSLA*, **SIGPLAN Notices**, **23(11)**, pp. 36-45, 1988.

[Symbolics86] *Symbolics Lisp Machine Manual*, Symbolics, Inc., Cambridge, MA, 1986.

[Xerox85] *Interlisp-D Reference Manual, Volume 3: Input/Output*, Xerox Artificial Intelligence Systems, Palo Alto, CA, 1985.