# SCIX

## A Scheme Interface to the X Window System

Håkan Huss
huss@nada.kth.se

Johan Ihrén
johani@nada.kth.se

March 30, 1990

**Abstract**

*SCIX* is an interface between the programming language Scheme and the X Window System. It is built entirely in Scheme apart from a few low-level routines that take care of the actual socket communication. By using Scheme on all levels, it is possible to design an interface that is more suited for applications written in Scheme than would be possible by linking with Xlib and using Xlib routines from within Scheme.

*SCIX* is completely object-oriented: all X resources, like windows, fonts and graphics contexts, are mapped onto Scheme objects and all communication is done through passing messages between these objects. This object-orientation is applied at the lowest level possible.

## Preface

*SCIX* was implemented at Digital Equipment AB, Sweden, and the Department of Computing Science at the Royal Institute of Technology (KTH), Stockholm, as the subject for our masters thesis at the School of Engineering Physics.

The development of the *SCIX* system would not have been possible without the help we received from Digital Equipment AB. They supplied us with DECstations to work on and gave us financial support so we could work full time (and more) on the system.

Digital has agreed to make the *SCIX* system publicly available under conditions resembling those of the X Window System from M.I.T.

# Contents

# 1 Technical Overview of the *SCIX* system

*Any sufficiently advanced technology*
*is indistinguishable from magic.*
*— Clarke's third law*

The *SCIX* system is a powerful environment for developing X applications in Scheme. It is completely object-oriented, in that all data is represented as procedure objects with local state. The objects are of two varieties, light weight and heavy weight. The light weight objects are small and are typically used where size and fast instantiation are more important than extensibility, as in color objects and graphic objects. The heavy weight objects constitute a powerful object system that supports multiple inheritance, and allows several objects to inherit the same instance of a superclass. It is also possible for a heavy weight object to inherit a light weight object.

*SCIX*'s goal was to make it possible to use the full expressibility of Scheme to manipulate a comprehensive model of X. This resulted in an extensible environment where whatever toolkit facilities are needed can easily be created in Scheme. A few demonstration widgets are implemented, but they are intended to show how widgets can be designed, rather than serve as a base for real world applications.

At the heart of the system there is an *X assembler*, which encodes requests in the protocol format, and an *X disassembler*, which translates incoming messages from the server to objects in the *SCIX* environment. The assembler resides as a method in the display object used for all communication with the window server. The disassembler is encapsulated in the *message handler*, an object in charge of all messages from the server. The message handler provides methods to manipulate server messages in various ways, including an empty message-dispatching loop.

Some of the features of the system are:

- It is easy to design new custom classes or subclasses from the *SCIX* core classes using the provided meta-constructor `define-class`.

- Because it is possible to inherit an *instance* of a class it proved useful to let the screen objects inherit their display. This led to fewer objects to manipulate and more concise code. The instance inheritance also facilitates general resource sharing.

- All windows support *callbacks*, which are procedures called by window objects given certain conditions.

- It is possible to explicitly control the mapping between a resource in the server and the manager of the resource in the *SCIX* environment. This is often used for windows that receive *events*. Usually the resource object handles these itself, but it is easy to have any procedure handle the events of an arbitrary group of resource objects.

- Error handling on a per-request basis is supported. This facilitates the use of very sophisticated debugging methods. It is possible to send an arbitrary object (eg., a continuation) along with any request as information that can be used by an error-handler. On a subsequent error the error-handler will be called with this object provided as an argument, otherwise it is automatically discarded.

- The object-oriented system supports dynamic method insertion, deletion and renaming. This means that objects can contain just the method packages they need. One example is a package of window manager communication methods that is touched upon briefly later.

In the rest of this report knowledge of Scheme is assumed. Excellent treaties on the language and its uses are [1] and [2]. As *SCIX* is designed directly upon the X Protocol, familiarity with X will help. In this paper the X Protocol denotes the core protocol of X version 11. Here the definite source of information is the X Protocol document in [3]. Some exposure to the concepts of object-oriented design is also useful, even if it is briefly covered in Appendix A.

# 2 Introduction and Rationale

*Language shapes the way we think*
*and determines what we can think about.*
*— B. L. Whorf*

On today's bitmapped graphic workstations it is possible to run several programs simultaneously. This requires a method of sharing the available screen resources. The most common method of sharing is through a *window server*. This is a program that runs on the same host as the display and controls what is output on the screen.

The application programs (often referred to as *clients*) display their output on the screen through requests to the server for resources. One consequence of the client communicating with a server (i.e., from a program) is that the client can execute on another host. This remote execution is called *network transparency* and is supported by the window server abstraction.

The communication between the client and the server is based upon a *protocol*. The communications protocol plus the server that implements it are usually called a *window system*. Note that the protocol defines the required capabilities of the server, as well as the format for requesting them. Therefore, the server is only an implementation of the server-side of the protocol. The two most well-known window systems today are *NeWS* [4], with a protocol based on PostScript and the *X Window System*, which uses the binary *X Protocol*.

The task of the server is to translate the incoming messages to graphics operations in the context of the current state of the screen and to dispatch messages informing clients what happens on the screen. The client translates its graphics requirements to the protocol format. It also translates the incoming messages (from the server) to some higher-level representation that makes sense in the environment of the application.

Routines that take care of the translations to and from the protocol format on the client side are usually available through special libraries. In the case of the X Window System a library of routines called *Xlib* provides an interface to the X Protocol from the C language [3]. Several toolkits are built on Xlib that provide higher-level abstractions for the primitives of the X Protocol. The *CLX* package does essentially the same thing as Xlib, but for Common Lisp [5]. There are several object-oriented toolkits based on CLX.

*SCIX* is an interface to the X Protocol for the Lisp dialect Scheme, just as Xlib is for C or CLX is for Common Lisp. The difference, apart from the target languages, lies mainly in their different approach to object-orientation.

One way to construct an interface between the X Window System and a programming language is to provide access to the Xlib-routines from within this other language. This has been done several times. However, for languages dissimilar to C, like Scheme, there are distinct advantages in not using Xlib. Scheme has much more powerful means of abstraction than C. This abstraction makes a more precise mapping possible between the Scheme environment and the protocol than in the C language. An interface designed entirely in Scheme can be smaller than Xlib for C is. Note that this does not mean that the actual code is smaller.

The object-oriented toolkits based on Xlib and CLX are based on imperative interfaces, *SCIX* is object-oriented from the lowest possible level. The *SCIX* interface is a bottom-up design, tightly coupled to the X Protocol. Its application layer is significantly closer to the protocol than interfaces constructed in a top-down fashion are. This is a definite strength as the logical design of X becomes apparent at the application level. A completely object-oriented system is easier to understand, easier to use. Here it provide a more comprehensive model of the window system. On the other hand, it is less efficient than an imperative system.

A window system is an ideal subject for object-oriented programming, as it consists of easily recognizable objects. Typically, a window system deals with independent windows coexisting and communicating with each other and the rest of the world. The windows may have subwindows, they may have colormaps associated with them and so on. An object-oriented approach fits this environment well.

A powerful, general purpose object-oriented programming environment is built on an encapsulated low-level interface to the X protocol. The low-level interface is a mapping of the protocol into Scheme.

Using an interpreted language like Scheme enables the user to test routines interactively, making it possible to develop different parts of an application independently. This capability is important in the area of graphics and user interface design. *SCIX* makes it possible to communicate directly and, most important, interactively with the X server. Application development is much easier when working interactively than when working in the usual edit—compile—run cycle. Understanding the window system model is much easier in an interactive environment with immediate, visual feedback. However, a purely interactive interface has severe limitations as it makes it impossible to create stand-alone applications. Furthermore, interpreted code normally is too inefficient for production use. If, however, the Scheme system allows compilation, these disadvantages disappear. *SCIX* was developed in the `Scheme->C` system [6], which provides both of these options.

# 3 The X Window System and Protocol

*I claim not to have controlled events,*
*but confess plainly that events have controlled me.*
*— Abraham Lincoln*

The X Window System is based on a few fundamental concepts. It is network transparent; the server may reside on one host and the clients on various other hosts. The clients could be running on machines with different architectures and operating systems.

Because clients may be anywhere on the network, it is impossible to communicate through files in a file system or via some sort of standard interprocess communication facility. Everything must go over the connection between the client and the server. Even in the case of client-to-client communication, communication must still flow through the server. Therefore the protocol designers have minimized use of network bandwidth. Often used data can be cached in the server, and most requests do not receive a reply. It is possible to make many changes to a window and then send them all as one request, for example.

In the protocol six different *resources* are recognized. These are *windows*, *pixmaps*, *fonts*, *colormaps*, *graphic contexts* and *cursors*.

A *window* is a rectangular area on the screen on which it is possible to draw graphics and text in different fonts. Windows may overlap and they may be mapped or unmapped (they may or may not be visible).

A *pixmap* can be described as an off-screen window. This means that almost anything one can do with a window one can also do with a pixmap. It can never be displayed, other than by copying the contents to a window. However, the contents cannot be destroyed by an overlapping window. So a pixmap is a "safe" place to compose something complicated, which is later copied out into a window with a fast raster operation.

A *colormap* can be thought of as a container for color cells. Each cell can contain any color, but only one at a time. It is possible for clients to allocate these cells in various ways for shared or non-shared use.

A *graphics context* is a network bandwidth reducing abstraction. It is a "drawing method" that is cached in the server and it contains information on which line-width to use when drawing lines, which colors to use, which font to use for text and several other things. Each graphic request only specifies the graphics context to use and thus avoids specifying all the details of that individual request.

There are several different types of messages possible between the client and the server. *Requests* are messages from the client to the server. There are two kinds: *one-way* and *round-trip* requests, the difference being that round-trip requests generates *replies* from the server. There are 120 requests defined in the core X protocol, 40 of which are round-trip.

X informs the clients of what the user does or what other clients does through *events*. There are many different types of events. To reduce bandwidth usage, the server only sends types of events the client explicitly has asked for on a per-window basis. For example, the client may receive an *Expose*-event when a previously hidden window or part of a window becomes visible. It may receive an *EnterNotify*-event when the cursor enters a window. After reception of an event, it is entirely up to the client to react on it. For instance, when a previously obscured part of a window becomes visible again the old contents are lost. But the client can choose to be informed of this through *Expose*-events and can thus keep the screen updated by repainting the areas it receives Expose-events for. If the window generating the *EnterNotify*-event acts as a button, then the client may highlight the button in some manner.

At any time, the user can do things outside the clients control, like iconifying or resizing a window or ignoring the application for a while. This flexibility leads to certain restrictions on the design of X applications. For instance, the clients should not make any assumptions about the size or position of its windows or what the user will do next—it should be able to cope with whatever situation develops. In consequence, a typical X application is *event-driven*, i.e., it does not require a certain sequence of events. The application sits in a main loop and reacts to events.

# 4 The *SCIX* Objects

*Any small object that is accidentally dropped*
*will hide under a larger object.*
*— from /usr/games/fortune*

From here on, we will use the term *objects* to refer to Scheme procedure objects. Messages are sent to these objects by calling the object with the message as the first, and possibly only, argument. For instance, the following example makes the workstation beep, provided that `dpy` is bound to a *display* object that has a connection with the X server:

```
(dpy 'bell 50)
```

Here `'bell` is the *message* sent to the object. The method associated with the message `'bell` requires one numeric parameter, indicating the percentage of base volume to use (50). All methods use a fixed number of parameters.

Most things in the *SCIX* environment are objects, but in complexity they span a great range: from simple *graphic* objects (like a polyline) that only honors a few methods to *display* and *window* objects that accept dozens of messages. It proved impossible to use the same underlying object implementation for all objects due to efficiency requirements. Another problem was that in object-oriented systems, objects are usually created by a dedicated *constructor*, a procedure that returns a new object of a distinct class. For instance the constructor `make-window` returns a new *window*-object. For reasons of efficiency, *SCIX* has generic reply, event and error constructors. That is, one constructor to construct all possible kinds of reply-objects, another for all events and a third for all errors.

These considerations led us to accept a heterogeneous system where the objects had two different underlying implementations: *heavy weight* and *light weight* objects.

## 4.1 Heavy Weight Objects

The *heavy weight* type provides the general object-oriented programming system (OOPS). This type is used primarily to implement objects that correspond directly to X resources. It is also the only type that supports inheritance, so any higher-level objects have to be based on the heavy weight object type (note, however, that it is possible to inherit *from* light weight objects). Constructors for new heavy weight object-classes are created by the macro `define-class`. For example, `define-class` is used below to create a primitive *button* constructor:

```
(define-class (primitive-button x y parent scr)
  (locals
   (button-state #f)           ; #f indicates 'off'
   (button-armed #f)           ; #t if had ButtonPress event in window
   ...)
  (inherit
   (window 12 12 (parent 'depth) x y parent 0
         'CopyFromParent 'CopyFromParent scr #f)); #f tells the window
                                                ; that it isn't top-level
  (methods
   (button-state (lambda () button-state))

   (scix-event-dispatch (lambda (event-source obj)
                          (let loop ((ev-s event-source))
                            ...
                            )))
   ...)
  (init
   (me 'scix-announce-id! me) ))
```

The result is a new constructor, `make-primitive-button`, which takes four parameters. $x$ and $y$ denote the location of the new button in the window *parent* on the screen *scr*. The body of the constructor-definition is in four parts: after the keyword `locals`, all local bindings are introduced. After the keyword `inherit`, superclasses to inherit are specified. Here the class `primitive-button` inherits the class *window*, and the parameters are those expected by the `make-window`-constructor. Note that it is possible to use arbitrary expressions as parameters; here the parent window instance (the parent window is already in existence) is queried for its depth, which is to be the depth of the button as well. After `methods`, are all local messages with their methods. Here, there is only the method associated with the message `button-state`, which returns the state of the button and the first few lines of the method corresponding to the message `'scix-event-dispatch` is shown. In reality, several more methods must inhabit the button to make it useful. The last clause, `init`, is followed by any Scheme forms to evaluate on instantiation. `'scix-announce-id!` is a method in the resource superclass that makes the existence of the object publicly known. `me` is a reference to the object itself, in this case the button. This announcement is necessary to enable the message handler (the routine that handles all incoming messages from the server) to find this particular object when, for instance, events arrive that this object should, or at least could, handle. This explicit control over the binding between the X resource identifiers (or *XIDs*) and the corresponding *SCIX* object is an unusual and extremely powerful mechanism.

So, a new button at $(10, 10)$ in the window $w$ is instantiated by

```
(make-primitive-button 10 10 w scr)
```

where `scr` is a reference to the screen object of the screen in use. There are many other possibilities within the general object system and they are all fully described in A and also in the reference manual supplied with the system.

## 4.2 Light Weight Objects

The *light weight* objects are small objects optimized for speed. This includes various graphic objects like polylines, polyrectangles, etc. This is the definition of the *polyline*-object:

```
(define-lw-class (polyline coord-mode point-list)
  (methods
   (draw (lambda (gc drawable . rest)
           (send-polyline drawable gc coord-mode
                          point-list (gc 'screen) rest) ))))
```

The macro `define-lw-class`, which creates new constructors of light weight objects, has exactly the same syntax as `define-class`, except for the `inherit`-clause, which it does not support. The polyline-object above has no local bindings and supports only one message, `'draw`, which takes three parameters, of which the third is optional. The first is a graphics context to use and the second is the drawable (that is, a window or a pixmap) to draw in. The third parameter, when present, is allowed to be anything at all. It is passed, unparsed, through the X assembler and will be available afterwards to handle any possible errors that this request may cause. As the code shows the message `'draw` to the polyline-object expands to a *PolyLine* request to the server. A polyline-object is created like this:

```
(define triangle
  (make-polyline 'origin '((0 0) (50 100) (100 100) (0 0))) )
```

The first parameter should be either `'Origin` or `'Previous`. The latter indicate that the coordinates of each point are relative to the previous point instead of relative to the origin of the window.

All server messages (that is, *replies*, *events* and *errors*) are represented by different light weight objects. There are a number of other light weight classes for different purposes, like the *atombox* object, which is a container for X protocol *atoms* residing within the display object. All the different classes and the messages they accept are described in the manual supplied with the system.

To make this environment inhabitable it was necessary to enforce some standards of behaviour between the objects. Therefore all objects support a few common messages to enable classification and to find the capabilities of encountered unknown objects. These universal messages are `'supported-messages`, which returns a list of all messages the object accepts, `'object-class`, which returns the object-class, `'me`, which returns a reference to the instance itself, `'object-system` which returns either `'heavy-weight` or `'light-weight` and, finally, `'object-desc`, which returns a description of the object and its methods that is to be used internally by objects that inherit this object, not by application code.

So, if `b1` is bound to the button created above and it is sent the message `'object-class`, that is:

```
(b1 'object-class)
```

the result will be

```
PRIMITIVE-BUTTON
```

If the message `'supported-messages` is sent to a polylineobject, this will be the result:

```
(DRAW)
```

as the graphics objects only support one method. If the same message is sent to a font object (created with `(make-font ‘‘fontname’’ scr)` a longer list will be returned:

```
(FONTNAME OPENFONT CLOSEFONT QUERYFONT QUERYTEXTEXTENTS ID
         SCIX-ANNOUNCE-ID! SCIX-DENOUNCE-ID!)
```

Note in these examples the absence of the messages `'object-class`, `'object-system`, `'object-desc`, `'supported-messages` and `'me`. These are excluded because they are always available, so there is no need to check for their existence.

# 5 The *SCIX* Environment

> *There was all kinds of mean nasty ugly people there on the bench... Chaosnet designers... Lisp hackers... TECO hackers. TECO hackers right there on the bench with me! And the meanest one of them, the hairiest TECO hacker of them all was coming over to me. And he was mean and nasty and horrible and undocumented and all kinds of stuff.*
>
> — *from Alice's PDP-10*

Because an X application is usually an event-dispatching loop, some other special facilities have proved useful. One can announce a procedure as the handler of a certain kind of event, usually on a per-window basis. Such a procedure is often referred to as a *callback*. Another facility allows the system to provide some sort of service to handle the incoming events, optionally while doing something useful. This facility is provided by the message handler in *SCIX*. Both these services are modeled on similar facilities in Xlib and Xt for C applications, see [3], [7] and [8].

## 5.1 Error Handling

A special construction provided in *SCIX* deals with error handling.

Here we present a short example of the error handling facilities available in *SCIX*. Every request directly mapped on an X request has an optional last argument, which is ignored by the X assembler. Should the request lead to an error, the error-object is paired with this last argument before any action is taken. The default error-handler merely notes the presence of such an argument. However, if the client provides an error-handler, by using the message `'install-error-handler`, the error-handler can use the argument in any way. If the request did not lead to an error then this *error object* is automatically discarded.

As the default error handler doesn't do anything but report the errors and the presence of an error object a new, more capable error handler is needed. Note that as usual the symbol `scr` is assumed to be bound to a screen object.

```
(define (install-new-error-handler scr)
  (letrec ((old-error-handler
            (scr 'install-error-handler
```

```
                    (lambda (error)
                      (let ((error-object (cadr error)))
                        (if (procedure? error-object)
                            (error-object error)
                            (old-error-handler error)))))))
    old-error-handler))
```

The new handler that the procedure above can install in the screen (or display) checks the error objects that come with the errors to see if they are procedures. If the error object is a procedure the error is dispatched to the error-object, otherwise the ordinary error handler is called.

The procedure `createpixmaps` below attempts to create a number of pixmaps from a list of descriptions. The intent is that even if some decriptions are faulty the other pixmaps should be created. The important thing is that the pixmap objects that correspond to faulty descriptions should be discarded automatically. This is achieved by supplying an error object with each invocation of the message `'createpixmap`. The error object, if called in the future, will simply remove the pixmap both from the list of pixmap objects and from the system table over announced resources.

```
(define (createpixmaps listofdesc scr)
  (let ((old-error-handler (install-new-error-handler scr)))
    (let ((pixmaps (map (lambda (desc)
                          (apply make-pixmap (append desc (list scr))))
                        listofdesc)))
      (for-each (lambda (pixmap)
                  (pixmap 'createpixmap
                          (lambda (error)
                            (set! pixmaps (remq! pixmap pixmaps))
                            (pixmap 'scix-denounce-id!))))
                pixmaps)
      (scr 'sync! #f)
      (scr 'install-error-handler old-error-handler)
      pixmaps)))
```

The method `'sync!` of the screen ensures that all requests have been precessed in the server so that, in this case, any errors that are generated are guaranteed to have arrived to the client before `'sync!` returns. The parameter specifies whether the method is allowed to discard any unprocessed input from the server. This is of course not acceptable, as we are want to take care of errors in a special way. Finally we try use defined procedure to create a list of pixmaps from a list of descriptions.

```
(define pixmap-data '((100 100 8) (100 100 12) (100 100 8) (-100 100 8)))

(length pixmap-data) ; -> 4

(define some-pixmaps (createpixmaps pixmap-data scr))

(length some-pixmaps) ; -> 2
```

On the system where this was done there was no support for drawables with a depth of twelwe pixels, therefore the second description was faulty. No X server will allow drawables with negative width, therefore the fourth description was faulty as well.

## 5.2 Callbacks

Callbacks are procedures that automatically get called when certain things happen, like the destruction of a window or the arrival of an especially interesting event type. The name "callback" is borrowed from the X Toolkit, see [8]. Callbacks are a powerful method of distributing responsibility for different situations to different routines once (typically when initializing the application), rather than each time they are encountered. Because of the asynchronous nature of events, this is important, as it is impossible to determine the state of the program when an event arrives.

In *SCIX*, all drawables and, for techical reasons, the object representing the keyboard can handle callbacks. If the application wants something done when a special kind of event occurs in a certain window, installing a callback in the window is the easiest way to do it. A callback in *SCIX* is a procedure called with two arguments when the event has occurred: the event object and the object where the event occurred. The callback can be considered as an extra event-handler for a specific event-class in a specific window or widget. For example, a callback that beeps when an *EnterNotify*-event occurs can be added to the window `w` like this:

```
(w 'add-callback! 'EnterNotify (lambda (event window)
                                 ((w 'screen) 'bell 50) ))
```

Here the callback procedure was defined on the spot, with `(lambda ...)`, but it could just as well be defined once and then installed as a callback procedure in several windows. Note that `'bell` is a method in the display object, but the screen object has inherited the display object instance and thus come to support the method. The same callback procedure could also theoretically handle different kinds of events, but it is not clear whether that would be useful in practice. It is also possible to remove installed callbacks in a similar fashion.

When an event arrives to the client, it is parsed by the *message handler*. The message handler produces an *event object* which accepts messages for selection of the data in the event. Because of the desire of the message handler to represent everything as objects, when appropriate, the reference to the window in which the event took place will be to the actual window object rather than just its id.

When the default event-handler receives the event, it extracts the reference to the window object and sends the message `'scix-event-dispatch` to that window with the event object as argument. When a standard window receives this message it dispatches the event to the callback for the event-class, if any, otherwise it just ignores it. A subclass to window can do whatever it wants both before and after executing the callback.

Here follows a short example of an application that handles *ButtonRelease*-events for two primitive buttons. The code for the primitive buttons is outlined in the section on the object-oriented system. If the first button is on after the *ButtonRelease*-event, a subwindow should be mapped. Otherwise, it should be unmapped. If the second button is pressed, the application will release all used resources and terminate. The code looks like this:

```
(define (demo-toggle scr)
  (let ((w    (make-window 200 200 8 0 0 (scr 'root) 0
                           'CopyFromParent 'CopyFromParent scr))
        (cmap (scr 'default-colormap)) )
    (let ((b1 (make-primitive-button 10 20 w scr))    ; "Map/unmap" button
          (b2 (make-primitive-button 10 40 w scr))    ; "Quit" button
          (w1 (make-window 50 50 8 130 70
                           w 0 'CopyFromParent
                           'CopyFromParent scr))   ; Subwindow
          (gc (make-gc (make-gc-value-mask `(foreground ,(scr 'blackpixel)))
                       scr))
          (red (cmap 'allocnamedcolor "red"))
          (t1 (list (make-imagetext8 50 70 "Some text"))) )
      (w  'createwindow (make-window-value-mask
                         `(background-pixel ,(scr 'whitepixel)) ))
      (w1 'createwindow (make-window-value-mask
                         `(background-pixel ,red)))

      (w  'add-callback! 'Expose (lambda (event window)
                                   (if (zero? (event 'count))
```

```
                                    (w 'draw t1 gc) )))

      (b1 'add-callback! 'ButtonRelease (lambda (event button)
                                           (if (button 'button-state)
                                               (w1 'mapwindow)
                                               (w1 'unmapwindow) )))

      (b2 'add-callback! 'ButtonRelease (lambda (event button)
                                           (if (button 'button-state)
                                               (begin
                                                 (w 'destroywindow)
                                                 (gc 'freegc) ))))
       (b1 'activate)
       (b2 'activate)
       (w 'mapwindow)
       (msg-handler 'mainloop
                    (lambda ()
                      (b2 'button-state) )
                    (list scr) ))))
```

The application is only interested in *ButtonRelease*-events in the primitive buttons. The buttons themselves also act on *ButtonPress*-, *EnterNotify*-, *LeaveNotify*- and *Expose*-events, but this is of no concern to the application. It should also be noted that the callbacks installed on the buttons should rather be handled automatically by the buttons. That is, a more powerful button might be instantiated and activated like this:

```
  ...
  (let* ((b1 (make-less-primitive-button x y w scr))
         (button-action (lambda ()
                          (...) )))
    ...
    (b1 'activate button-action)
    ...
```

Note that we only destroy the top window `w` is destroyed, as the buttons and subwindows automatically are destroyed when the parent is. *SCIX* automatically maintains internal consistency.

## 5.3 The Main Program

In a pure event-driven loop, the type of the event that arrives determines what will happen next. This model does not satisfy the needs of all applications. For instance, the application could require some calculation and respond to user interactions simultaneously. This is not easy to achieve because while the calculation progresses, anything the user does only results in events being queued, but the application will not find the events until it looks for them.

This is a situation where one might use an implementation specific interrupt scheme to satisfy the conflicting demands on the process. Interrupts in this context are *signals* that cause a certain routine, the *interrupt handler*, to be executed when new data arrive to the client. The program halts instantly and execution is transferred to the interrupt handler, which typically parses the new data, acts on the events and then terminates, whereupon the program resumes its execution exactly where it was.

The problem with an interrupt driven design is that different kinds of data can arrive. *Events* are, by nature, asynchronous and usually lead to the invocation of a callback installed in a window. This is perfectly possible to do in an interrupt routine. The program waits for *replies*, however. Therefore the interrupt handler cannot take care of replies.

What *SCIX* provides for this situation, instead of an interrupt-driven main loop, is not as elegant, but powerful enough. The application can use the `'mainloop` method in the message handler to handle events while, optionally, simultaneously proceeding with a background calculation. An invocation could look like this:

```
(msg-handler 'mainloop terminate?
                       (list scr1 scr2)
                       (lambda (handle-events)
                         (let loop ()
                           (calculate)
                           (handle-events)
                           (if (not (finished?))
                               (loop) ))))
```

The first argument, `terminate?`, is a procedure with no arguments provided by the application that returns `#t` when the main loop should terminate. This procedure is actually only used when no third argument is provided. The second argument is a list of screen objects indicating the servers the main loop will watch for events. If no third argument is present, `mainloop` will sleep until input is available on one of the connections. Then it will dispatch on the events, and finally check for termination with a call to `terminate?` before going back to sleep. The third argument, when present, should be a procedure of exactly one argument (which is bound to an event-handling procedure by the message handler) which does anything for as long as it wishes and calls the argument procedure whenever it wants to handle possible queued events. When the calculation terminates, the main loop also terminates.

### 5.4 Toolkit Design

As the X Window System intentionally "provides only mechanism and not policy", the field is free for toolkit designers to add higher-level abstractions to the interfaces to X. The plethora of widget libraries that build upon the *X Intrinsics*, that in its turn is built upon Xlib are good examples in the C language.

What toolkits provide are more object-oriented, higher-level representations of the window system with both new abstractions, like menus, buttons and scrollbars and new mechanisms, like more powerful queue-handling functions and *callbacks*.

Because *SCIX* is written in Scheme, it is already capable of handling lists and queues efficiently. There is also a quite general callback facility built into all windows. What *SCIX* lacks is a library of *widgets* (a common term for higher-level derivatives of windows, like menus and buttons of various kinds). However, it is not hard to design widgets. For example of this, appendix C contains a sample implementation of a button hierarchy.

### 5.5 Other Means of Abstraction

The *SCIX* environment gains a lot of power from its object-oriented subsystem. This subsystem enables the system to offer a number of powerful constructs to the user. Scheme plus object-orientation provides a much richer set of elegant expressive abstractions that can be fully exploited within the *SCIX* system than is possible in the C language. Since the system has only recently become stable enough to allow application design, it is still to early to say exactly what will prove useful.

One thing that is currently lacking is a mechanism for client-to-client communication through properties. A prototype implementation that provides these services through the new methods `'getprop` and `'putprop` shows promise. An immediate use for this will be to enable *SCIX* clients for conform to the ICCCM [9].

The use of higher-level constructors is also interesting. Higher-level constructors are procedures that use the specialized constructors to instantiate a number of objects, with or without additional management of those objects. For example, the primitive button constructor shown earlier is not very general. It creates two internal graphics contexts (also called *gc's*) in each button. A more elegant version would instead take the gc's as parameters and thus share them between a number of primitive buttons. However, binding these gc's in the main program constitutes a breach in the abstraction barriers (the main program is interested in buttons, not any gc's they happen to use). Here a higher-level constructor could encapsulate the primitive button constructor and create the two gc's itself before instantiating a number of buttons using these gc's.

Other possibilities include general widget managers. I.e., it is possible to announce any procedure object as the destination for events from an arbitrary group of windows/widgets by simply instantiating them as non-top-level and then announcing another procedure as the manager of everything concerning these windows/widgets. It is then left to the managers discretion whether it dispatches events to the windows or handles them entirely on its own.

# 6 Examples

*If the code and the comments disagree,*
*then both are probably wrong.*
*— Norm Schryer*

In all the examples in this report, the main procedure always takes one parameter, `screen`, which is a reference to a screen-object. This is not a restriction. Instead, it is a consequence of the interactive nature of Scheme programming. After Scheme is started, a connection to the local X server is obtained like this:

```
(define dpy (make-display ":0"))
```

(The reason for using the name `dpy` instead of the more logical `display` is that `display` is a Scheme standard procedure.) Here the display specification `":0"` indicates that a connection to the "first" (and usually only) X server on the same host is wanted. Now `dpy` is bound to a display object that supports a large number of messages. In *SCIX*, however, the display object is seldom used directly. Instead, it is accessed through a screen object that has inherited the display object *instance*. That is, if the workstation has several physical screens, then *SCIX* will have several screen objects, but they will all inherit the *same* display object. The default screen object is obtained through the `'defaultscreen`-message to the display object. That is

```
(define scr (dpy 'defaultscreen))
```

binds the symbol `scr` to the screen object that is usually used. *SCIX* allows using these two objects, `dpy` and `scr`, for running different applications that all use `scr` as a parameter rather than requiring every new application (in the same Scheme) to open a new connection to the X server. Of course, when creating stand-alone applications, this practice is no longer useful. In `Scheme->C` a procedure compiled as the entry-point to an application will be called with one argument, which is a list of strings representing the command-line arguments with which the application was invoked (the equivalent of `argv` in C). Then something like this will be used instead:

```
(define (stand-alone-application clargs)
  (let* ((dpy (make-display (cadr clargs)))
         (scr (dpy 'defaultscreen)) )
    (application scr) ))
```

This is a very primitive solution that assumes that the second command-line argument (that is, the first after the program name) is an X server specification. A real application needs a more sophisticated solution.

## 6.1 Hello World!

By tradition, the now famous "Hello World!"-example often is used to display the mechanics of new interfaces to the X Window System. See [10] for a C version. Here it is in a slightly enhanced *SCIX* reincarnation. The enhancement lies in the handling of *Expose*-events.

```
(define (demo-hello scr)
  (let* ((w (make-window 400 400 (scr 'root-depth) 0 0 (scr 'root)
                         0 'CopyFromParent 'CopyFromParent scr))
         (f1 (make-font "8x13bold" scr))
         (view (make-view))
         (gc (make-gc (make-gc-value-mask `(foreground ,(scr 'blackpixel))
                                          `(background ,(scr 'whitepixel))
                                          `(font ,f1) )
                      (scr 'root)))
         (hello (make-imagetext8 100 50 "Hello World!"))
         (keybd (scr 'keyboard))
         (terminate #f) )

    (view 'add! (list hello))
    (w 'add-callback! 'Expose (lambda (event window)
                                (if (zero? (event 'count))
                                    (begin
                                      (w 'draw (list view) gc)
                                      (scr 'flush!) ))))
    (w 'add-callback! 'ButtonPress
       (lambda (event window)
         (let ((x (event 'event-x))
               (y (event 'event-y)))
           (let ((hi (list (make-imagetext8 x y "Hi!"))))
             (view 'add! hi)
             (w 'draw hi gc)
             (scr 'flush!) ))))

    (w 'add-callback! 'KeyPress
       (lambda (event window)
```

```
        (let* ((key (event 'detail))
               (keysym ((keybd 'getkeyboardmapping key 1) 'keysyms)) )
          (if (= (char->integer #\Q) (caar keysym))
              (begin
                (w 'destroywindow)
                (gc 'freegc)
                (f1 'closefont)
                (scr 'flush!)
                (set! terminate #t) )))))
    (w 'createwindow
       (make-window-value-mask `(background-pixel ,(scr 'whitepixel))
                               `(event-mask
                                 ,(make-event-mask 'ButtonPress
                                                   'KeyPress
                                                   'Exposure))))
    (w 'changeproperty 'replace 'wm_name 'string 8 "SCIX Hello World!")
    (w 'mapwindow)
    (scr 'flush!)
    (msg-handler 'mainloop (lambda () terminate) (list scr)) ))
```

The "Hello World!" program puts up a white window on the screen and writes the string `"Hello World!"` in it. When a button is pressed the string `"Hi!"` is written at the current coordinates of the pointer. When the `Q`-key is pressed the program exits.

First all needed resources are created. Then a termination-predicate, `terminate?`, is defined. It is later used by `mainloop` to determine when to quit. Then three callbacks are installed in the window, followed by the window being created in the server and mapped. Note the difference between the creation of the *SCIX* object and the creation of the resource in the X server. After that the program enters the `mainloop`-routine that handles the event-dispatches to the callback procedures. Any *Expose*-events lead to a redraw of everything that has been written before. This is handled through the use of a view-object. A view-object is a wrapper around a number of graphic objects that can be used exactly as any other graphic object. It can be noted that this handling of *Expose*-events is very primitive. There are several alternative methods. Another observation is that to use the method `'changeproperty` directly is the most primitive way of interacting with the window manager. The proper way to do it is possibly something like

```
(w 'putprop 'wm_name "SCIX Hello World!")
```

as has been discussed in a previous section.

## 6.2 Color Wheel

The idea behind the next example is borrowed from a Digital Equipment demonstration program. This application allocates all available color cells. Then it draws a "color wheel" made out of as many triangles as color cells, each in a different color. Then the wheel rotates until a button is pressed in the window containing the wheel.

```
(define (demo-wheel size scr)
  (let* ((cmap (scr 'default-colormap))
         (w    (make-window size size (scr 'root-depth) 0 0 (scr 'root)
                            0 'CopyFromParent 'CopyFromParent scr))
         (gc (make-gc (make-gc-value-mask `(foreground ,(scr 'whitepixel)))
                      (scr 'root)))
         (colors '())
         (radius (inexact->exact (round (/ size 2))))
         (terminate? #f) )

    (set! colors (allocate-all-colors cmap))
    (if (> (length colors) 2)          ; It is POSSIBLE to get by with 3.
        (begin
          (format #t "Using ~a colors.~%" (length colors))
          (w 'createwindow (make-window-value-mask
                            `(background-pixel ,(scr 'blackpixel))
                            `(event-mask ,(make-event-mask 'ButtonPress)) ))

          (w 'add-callback!             ; Terminate on ButtonPress
             'ButtonPress
             (lambda (event window)
               (cmap 'freecolors (map (lambda (color)
                                        (color 'pixel))
                                      colors)
                     0)
               (gc 'freegc)
               (w 'destroywindow)
               (scr 'flush!)
               (set! terminate? #t) ))

          (w 'changeproperty 'replace 'wm_name 'string 8 "SCIX Color Wheel")
          (w 'mapwindow)
          (scr 'flush!)
          (load-hls-wheel colors cmap)
          (draw-wheel w gc colors radius)
          (scr 'flush!)
          (msg-handler 'mainloop
                       #f
                       (list scr)
```

```
                       (lambda (handle-events)
                         (rotate-colors scr w cmap colors
                                        (lambda () terminate?)
                                         handle-events) )))
        (begin
          (format #t "Could not allocate enough colors.~%")
          #f))))

;;; allocate-all-colors -- returns a list with as many color objects
;;;                        as it was possible to allocate r/w. We
;;;                        start by grabbing them 15 at a time, but
;;;                        the last ones are allocated one at a time.
;;;                        Note that the allocated colors will only
;;;                        have the pixel-values set correctly, the
;;;                        rgb values will be undefined.
(define (allocate-all-colors cmap)
  (define (create-colors reply)
    (if (pair? reply)
        (map (lambda (pixel)
               (let ((c (make-color 0 0 0)))
                 (c 'set-pixel! pixel)
                 (c 'set-colormap! cmap)
                 c))
             (caddr reply) )
        reply))

  (let ((colors '()))
    (let loop ((colors-at-a-time 15)
               (c (create-colors (cmap 'alloccolorcells #f 15 0))))
      (if (pair? c)                      ; A list of new color objects
          (begin
            (set! colors (append colors c))
            (loop colors-at-a-time
                  (create-colors
                    (cmap 'alloccolorcells #f colors-at-a-time 0) )))
          (if (not (= 1 colors-at-a-time))        ; Try to get a few more
              (loop 1 (create-colors (cmap 'alloccolorcells #f 1 0)) ))))
    colors))

;;; load-hls-wheel -- loads a HLS wheel into the color objects.
(define (load-hls-wheel colors cmap)
  (let ((denom (+ (length colors) 1)) (rgb #f))
    (let loop ((i 0) (color (car colors)) (colors (cdr colors)))
      (set! rgb (hls->rgb (list (/ (* i 360) denom)
                                0.5
                                0.5)))
      (color 'set-red! (* 65535 (car rgb)))
```

```
      (color 'set-green! (* 65535 (cadr rgb)))
      (color 'set-blue! (* 65535 (caddr rgb)))
      (if (pair? colors)
          (loop (+ i 1) (car colors) (cdr colors))
          (cmap 'storecolors colors) ))))

;;; hls->rgb -- returns a RGB triplet from the input HLS ditto. The actual
;;;             code is of no interest here.
(define (hls->rgb hls)
  ...)

;;; draw-wheel -- draw a wheel made up of as many triangles as there are
;;;               colors available and fill each triangle with one color.
;;;               It is possible to write this without using set!, but
;;;               will be harder to read then.
(define (draw-wheel w gc colors radius)
  (let ((tri #f) (dphi (/ (* (/ 360 (length colors)) 3.141592) 180))
                 (center (list radius radius)) (pt1 #f) (pt2 #f) )
    (let loop ((phi 0) (colors colors))
      (if (not (null? colors))
          (begin
            (set! pt1 (list (float->fixed (+ radius (* radius (cos phi))))
                            (float->fixed (+ radius
                                             (* radius (sin phi)))) ))
            (set! pt2 (list (float->fixed
                             (+ radius (* radius (cos (+ phi dphi)))) )
                            (float->fixed
                             (+ radius (* radius (sin (+ phi dphi)))) )))
            (set! tri (make-fillpoly 'convex 'origin `(,center ,pt1 ,pt2)))
            (draw-with-color w gc (list tri) (car colors))
            (loop (+ phi dphi) (cdr colors)) )))))

;;; rotate-colors -- rotates the rgb values in the list of color objects.
;;;                  The arguments terminate? and handle-events are there
;;;                  to make this routine conform with the requirements of
;;;                  the calc parameter of mainloop.
(define (rotate-colors scr w cmap colors terminate? handle-events)
  (let ((save-pixel #f))
    (let loop ()
      (set! save-pixel ((car colors) 'pixel))
      (let iloop ((colors colors))
        (if (not (= 1 (length colors)))
            (begin
              ((car colors) 'set-pixel! ((cadr colors) 'pixel))
              (iloop (cdr colors)) )))
      ((car (last-pair colors)) 'set-pixel! save-pixel)
      (cmap 'storecolors colors)
```

```
      (scr 'flush!)
      (handle-events)
      (if (not (terminate?))
          (loop) ))))

;;; draw-with-color -- changes the foreground-color of gc to color before
;;;                    drawing the graphic objects in graphic-l
(define (draw-with-color w gc graphic-l color)
  ((gc 'mask) 'set! `(foreground ,color))
  (gc 'changegc (gc 'mask))
  (w 'draw graphic-l gc) )
```

This application does not show off *SCIX* capabilities in a fair way, because the mainloop consists of two messages to color objects (**'set-pixel!** and **'pixel**) for each color and the **(cmap 'storecolors colors)**-call will have to use four messages to the color objects (one each for pixel-value and red-, green- and blue-values) for each color when translating it to X Protocol format. If the wheel consists of 250 colors this will lead to 1500 messages to the color objects per step. Timing the wheel on a DECstation 2100, this indicates that about 6500 messages are handled per second in addition to everything else that is done. All messages discussed above are messages between *SCIX* objects, not messages to the X server.

A more elegant program design might be found by changing the primary data structure. Instead of maintaining and mutating a linear list of colors, let each color index into a circular list of pixel-values. This circular list then enables us to rotate by simply advancing the list base by one step.

The problem of this design is that the assembling function has to be aware of the new method of obtaining the appropriate pixel-value. Using inheritance, this is easy. To provide an example of multiple inheritance, the pixel-list is encapsulated in a new object:

```
(define-lw-class (pix-container)
  (locals (pixels '()))
  (methods
   (add-pixel! (lambda (pix)
                 (set! pixels (cons pix pixels)) ))
   (make-circular! (lambda ()
                     (set-cdr! (last-pair pixels) pixels) ))
   (pixel-ref (lambda (n)
                (list-ref pixels n) ))
   (rotate! (lambda ()
              (set! pixels (cdr pixels)) ))))
```

The `pix-container` object now can handle the pixels adequately. To ensure that the color objects use this structure, create a new color object (`my-color`). This new object inherits a color object and a `pix-container` object, and overrides the standard method of accessing pixels by a local `pixel`-method.

```
(define-class (my-color n pix-list color)
  (inherit pix-list color)
  (methods
   (pixel (lambda ()
            (me 'pixel-ref n) )))
  (init
   (me 'add-pixel! (me '(color pixel))) ))
```

The remaining code will have to be changed as follows. In the initial `let*` binding, add

```
(the-pixels (make-pix-container))
```

and before drawing the wheel ensure that the list is circularized. The rotation of the colors now is simple enough to write the call to `mainloop` thus

```
(msg-handler 'mainloop
             #f
             (list scr)
             (lambda (handle-events)
               (let loop ()
                 (the-pixels 'rotate!)
                 (cmap 'storecolors colors)
                 (scr 'flush!)
                 (handle-events)
                 (if (not terminate?)
                     (loop) )))))
```

The major change is to `allocate-all-colors`, since this function now also has to add all the pixels to the pixel list:

```
(define (allocate-all-colors cmap pixels)
  (define (create-colors reply)
    ...)

  (let ((colors '()))
    (let loop ((colors-at-a-time 15)
               (c (create-colors (cmap 'alloccolorcells #f 15 0))) )
```

```
   (if (pair? c)                      ; A list of new color objects
       (begin
         (set! colors
               (append colors
                       (let iloop ((n (length colors))
                                   (col-list c)
                                   (result '()))
                         (if (null? col-list)
                             result
                             (iloop
                              (+ n 1)
                              (cdr col-list)
                              (append result
                                      (list
                                       (make-my-color
                                        n
                                        pixels
                                        (car col-list) ))))))))
         (loop colors-at-a-time
               (create-colors
                (cmap 'alloccolorcells #f colors-at-a-time 0) )))
       (if (not (= 1 colors-at-a-time))         ; Try to get a few more
           (loop 1 (create-colors (cmap 'alloccolorcells #f 1 0)) ))))
 colors))
```

One may be lead to think that this is a more efficient implementation than the first, as the rotating of pixels is done in constant time. Unfortunately, the `list-ref` operation is linear in time, making the lookup of all pixels in the `'storecolors` $O(n^2)$. Since the original program used a rotation operation which is $O(n)$, and the lookup of pixels also is $O(n)$, the first program runs in linear time, whereas the second is quadratic. In order to remedy this, one would have to adopt an alternative storage strategy, eg., using a vector rather than a list to store the pixels. This is easy to do and would only affect the definition of `pix-container`. The `demo-wheel` supplied with the SCIX system is implemented in this way.

# 7 Implementation

*Here be dragons!*
*— ancient map*

An X request is normally generated indirectly when a message is sent to an object. This involves several phases or “layers”. Following a specific request is the easiest way to illuminate this. Assuming that **cmap** is bound to a created colormap object, sending the message **'allocnamedcolor** to allocate, for instance, the color yellow, i.e, issuing

```
(cmap 'allocnamedcolor "yellow")
```

will ultimately result in the request *AllocNamedColor* being sent to the server. Then the reply (which contains information about rgb-values and the pixel value for the allocated color) will be taken care of and a *color object* will be returned as the result of the operation.

In the colormap object, this is the beginning of the code associated with the message **'allocnamedcolor** (the rest will follow):

```
(allocnamedcolor (lambda (name . rest)        ; #85: AllocNamedColor
                   (let ((reply (send-allocnamedcolor me name scr rest)))
                      ...))))
```

So the method expands essentially to a let-binding of **reply** to whatever the result of **send-allocnamedcolor** is. The **me** in the call to send-allocnamed-color is a reference to the colormap object itself and **scr** is a reference to the screen object representing the screen of the actual colormap. **rest** is an unparsed object that can be supplied as a per-request error-handler, as described elsewhere. The **send-allocnamedcolor** function looks like this:

```
(define (send-AllocNamedColor cmap name scr . rest)
  (let* ((n (string-length name))
         (seq-nr (scr 'scix-xas `((,a-request  . ,AllocNamedColor)
                                  (,a-card8    . 0)  ; 1 unused byte
                                  (,a-card16   . ,(+ 3
                                                     (/ (+ n (pad n)) 4)))
                                  (,a-colormap . ,cmap)
                                  (,a-card16   . ,n)
                                  (,a-card16   . 0)  ; 2 unused bytes
                                  (,a-string8  . ,name) )
                      rest)))
    (msg-handler 'wait-for-reply seq-nr 'AllocNamedColor scr) ))
```

Here the step from the object-oriented layer down to the X Protocol layer is taken. This code composes an encoding of the *AllocNamedColor* request which is then sent on to the *X assembler*, `scix-xas`, which is contained in the display object. Here is the code for the *scix-xas*-method in the display-object:

```
(scix-xas (lambda (l rest)
            (let ((current-request-size 0))
              (zero-buffer-position)
              (for-each (lambda (p)
                          ((car p) (cdr p) current-request) )
                        (cdr l) )
              (set! current-request-size (current-buffer-position))
              (if buffer
                  (if (> (+ tot-buffer-size current-request-size)
                         max-buffer)
                      (begin                ;; Need to flush the buffer
                        (writesock fd request-buffer tot-buffer-size)
                        (set! tot-buffer-size
                              (string-append! request-buffer
                                              current-request
                                              0
                                              current-request-size)) )
                      (begin
                        (set! tot-buffer-size
                              (string-append! request-buffer
                                              current-request
                                              tot-buffer-size
                                              current-request-size)))))
              (begin
                (writesock fd current-request current-request-size)
                (set! tot-buffer-size 0) ))
            (set! seq-nr (+ 1 seq-nr))
            (let ((r (strip-object rest))) ; Any special error object?
              (if (not (null? r))
                  (set-cdr! (last-pair history-list)
                            (list (cons seq-nr r)) )))
            seq-nr)))
```

The encoding is evaluated according to its type, and stored in a buffer in the display object, where the data is kept until it is actually sent to the server. Most of the code above deals with this buffering.

Usually everything is buffered until either flushing is specifically requested (using the message `'flush!` of the display object), or the application attempts to read from the server. `writesock` is a C routine that writes out the bytes on the socket referred to by the socket descriptor `fd`.

We have followed the request all the way down from the message `'alloc-namedcolor` to the colormap object to the corresponding bytes being written to a socket. Now it is time to examine what happens to the reply from the server.

In the code for `send-AllocNamedColor` above one finds that after sending the message `'scix-xas` to the screen object there is a call to the `wait-for-reply` method of the message handler:

```
...
    (msg-handler 'wait-for-reply seq-nr 'AllocNamedColor scr)  ))
```

The `wait-for-reply` method does just that: it reads the message stream from the server synchronizing on the sequence number, because when the reply arrives (or an error if something has gone wrong) it will be tagged with the sequence number of the corresponding request. Here is the code:

```
    (define (wait-for-reply seq-nr reply-type dpy)
      (if handling-event
          (mutex-wait-for-reply seq-nr reply-type dpy)
          (let loop ((found-msgs (next-message dpy reply-type)))
            (if (not found-msgs)
                ;; Did not find a message
                (loop (next-message dpy reply-type))
                ;; Found a message
                (let ((r (find-msg-with-seq-nr seq-nr reply-type)))
                  (if (not r)           ; Didn't find it, sorry
                      (loop (next-message dpy reply-type))
                      r))))))
```

The `mutex-wait-for-reply` procedure is used when the calling procedure is an event handler, and is almost identical to this one. The difference is, that while `wait-for-reply` dispatches on events having arrived before the reply came, `mutex-wait-for-reply` doesn't.

`find-msg-with-seq-nr` is a procedure that dispatches on every message queued until a reply or error with the correct sequence number is found. This is then returned as the value.

`next-message` is a routine that reads from the socket with `readfrom-server`, then applies the *X disassembler,* `xdas` on the resulting raw data (a byte string). The code for `next-message` (with code to handle incomplete reads omitted):

```
(define (next-message dpy reply-type)
  (dpy 'flush!)
```

```
  (non-blocking-io (dpy 'scix-socket))
  (let loop ((str (readfromserver dpy)))
    (if (null? str)
        (begin
          (blocking-io (dpy 'scix-socket))
          #f)                                ; No, did not find anything
        (begin
          (let ((r (xdas str reply-type dpy)))
            ...
            (set! msg-list (append msg-list (list r))) ; Queue parsed msg
            (if (not (end-of-input-string? str))
                (loop str)                         ; Parse rest of string
                (begin
                  (blocking-io (dpy 'scix-socket))
                  #t)))))))                      ; Yes, did find something
```

The routines `blocking-io` and `non-blocking-io` are written in C. The former enables blocking read from the socket, the latter disables it.

`xdas` interprets the raw data and returns objects as its result. To create objects, it uses the three object-constructors `make-reply`, `make-event` and `make-error`. Here is the code (again, without the checks for incomplete reads):

```
(define (xdas str reply-type dpy)
  (let* ((type (get-next-byte! str))
         (masked-type (bit-and type #x7f)) ) ; Mask off SendEvent-flag
    (cond ((zero? type)                  ; This is an error
           ...
           (make-error str dpy) )
          ((= type 1)                    ; This is a reply
           ...
           (make-reply reply-type str dpy) )
          ((and (>= masked-type 2) (<= masked-type 34)) ; This is an event
           ...
           (make-event type str dpy) )
          (else
           (error 'xdas "Unknown type of message: ~a." type) ))))
```

Now we either have a reply-object or an error-object and this is returned backwards through the calling sequence to the colormap-object, where `reply` (within the *allocnamedcolor*-method) gets bound to this object. Here is the code for the part of the method omitted above:

```
...
(let ((reply (send-allocnamedcolor me name scr rest)))
```

```
(if (x-reply? reply)
    (let ((c (make-color (r 'exact-red)
                         (r 'exact-green)
                         (r 'exact-blue)))
          (pixel (r 'pixel)))
      (c 'set-visual-red! (r 'visual-red))
      (c 'set-visual-green! (r 'visual-green))
      (c 'set-visual-blue! (r 'visual-blue))
      (c 'set-name! name)
      (c 'set-pixel! pixel)
      (c 'set-colormap! me)
      c)
    #f))
```

Now, if it is a reply then everything is fine and a color object is created and subsequently returned, but if it is an error then `#f` is returned instead, indicating failure.

This is about as complicated as a single request ever gets. Most requests are not round-trip, so they just execute the assembler part of the above code. Also, for round-trip requests the reply object is usually returned as it is, leaving to the application to handle it. In this example, however, it is natural to return a color object as a color was requested.

# 8 Implementation Status

*The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time.*

*— Tom Cargill*

In the current implementation, there are areas of the protocol that are handled—that is, the requests are accessible through messages to objects—but we have not had the time to test them properly. Therefore, *SCIX* probably doesn't handle some parts of the protocol optimally. More specific limitations:

- Practically no support for *keysyms* and *keycodes* is provided. No translation tables, etc., are implemented, although the mechanisms are all there.
- *Extensions* to the core protocol are not handled at present. The main reason for this is that there were no extensions in the available X servers until recently, so we had nothing to experiment with. However, we are currently experimenting with different methods of handling extensions and it looks like it will be possible to handle it in a very nice way.
- *Properties* and *selections* are handled, but in a very primitive way. We are playing with nicer ways of manipulating these though.

It should be noted that the limitations mentioned should all be easy to correct, but we have chosen to leave them out rather than to implement them incorrectly as we did not have the time to evaluate which solution would be best.

# 9 Performance Issues

*Real Programmers write self-modifying code,*
*especially if they can save 20 nanoseconds in the*
*middle of a tight loop.*

*— from Real Programmers Don't Use PASCAL*

The *SCIX* system does not have anything near the event-handling capability of a typical Xlib application. For example, the color wheel example shown previously is about three times slower than the Xlib-based program it is derived from.

In pure Scheme it is very difficult to achieve really good performance for various reasons.

If portability is sacrificed on the altar of efficiency, it is probably possible to speed up the disassembler substantially. In that case, instead of parsing the entire message, field for field, copy the message as is into storage within the object. Parsing each field once, when it is referenced for the first time, for example using memoization mechanisms. Because most messages from the server are queried only for one or a few fields, this would lower the disassembler overhead.

There is also additional performance to be gained from tuning the object oriented system and compiling low-level functions without type and bounds checking.

Another problem is the size of the generated applications. Currently the size of a non-optimized typical application like the color wheel is about 1.5 megabytes on a RISC-machine like the DECstation. This is very large, compared to typical X applications but easy to understand. As the X requests are grouped as methods to a few core classes in *SCIX*, almost all the request code will have to be present even though only a few methods in the objects are used. This is very hard to do something about, but in an environment that uses dynamic loading this will not be as important as it may seem.

# 10 Conclusions

*Anything that works is outdated.*
*— Ain Ala*

The *SCIX* system provides a very powerful environment for interactive developement of X applications. As it is implemented enetirely in Scheme it can probably never be as fast a C based applications. However, *SCIX* does provide a nice interface that shows the logical design of the X Window System.

The system is very coherent, i.e., the manipulation of low-level objects is done in precisely the same manner as for high-level objects, like buttons. This consistency in style makes it easy to grasp the model of X provided by *SCIX*. Also, this approach avoids the gap between the X interface and the higher-level toolkits apparent in Xlib–Xt.

*SCIX* is useful as a graphical interface for Scheme. It is also an ideal tool for understanding window systems in general and X in particular.

# 11 Acknowledgements

During the development of the *SCIX* system we have not had time for very much else. This was especially true during the final stages. Therefore, we want to acknowledge the help and support we have received from friends and family. We also want to thank Richard Hallin at Digital Equipment AB, a man with magic fingers. At times we would not have managed without his help. Olof Gullnäs, Digital, helped us organize our thoughts by asking the critical questions in times of confusion (it definitely *was* confusing at times).

At KTH we want to thank Joacim Halén, our advisor, who has a well organized mind and is willing to share. We also want to thank Joel Bartlett at DECWRL, both for providing us with the eminent `Scheme->C` system and also for answering our questions in times of need.

We also would like to express our endless gratitude to Jane Boreta, Digital, who went to great lengths to proofread this report. The readable parts are clearly due to her. All remaining faults are ours.

Lastly, we want to thank Magnus Persson, Digital, whose original idea developed into the *SCIX* system. He tried hard to find time for many design discussions with us in his tight schedule. He also did most of the implementation of the object-oriented system employed in *SCIX*, thereby enabling us to concentrate on the X side.

# A The Object-Oriented Programming System

*It is much easier to use macros*
*than to define them.*
*— Peter Brown*

by Magnus Persson

## A.1 Introduction

This appendix defines an object-oriented programming system in Scheme, Heavy-Weight OOPS, or *HWOOPS*. The term heavy-weight is a pun upon the fact that objects in the system will coexist with less general, or lighter, objects. That is, a class in HWOOPS may inherit a class in another system by using the appropriate protocol.

The notion of object and (meta-) class in object-oriented programming is a bit confusing. In a richly typed language like Scheme, it is fairly easy to define the concepts. In the rest of this appendix, an object will be called an instance object and a (meta-) class will be called a (meta-) class object. These objects will always possess the type $A \to B$; where $A$ is the argument type and $B$ is the result type—thus, instance and (meta-) class objects will always be function—or, in Scheme—procedure objects. This distinguishes them from the more general notion of objects or values in Scheme.

A class object has an arbitrary argument type and an instance object result type. An instance object has a finite argument type (usually a token subtype) and an arbitrary result type. A meta-class object has an arbitrary argument type and a class object result type.

The elements in the instance object argument type are usually called *messages* or *method names*. Since this type is finite, the instance object itself is finite (each pair is called a *method*). An application of this function object is called *message passing*. By using a concept called *inheritance*, the instance object is structured in different ways (usually described by a graph), thus enabling different interpretations in different contexts (see below).

## A.2 The Definition of HWOOPS

The purpose of the system is not to provide the world with yet another object-oriented system but to provide a usable system for the interface between Scheme and X. The entire source code for HWOOPS is written in Scheme. Beside some convenience routines, it is fully contained in the `define-class` macro (implemented using the *extend-syntax* mechanism, see [1]).

HWOOPS supports most of the common object-oriented programming features—like multiple inheritance, sharing of superclass instance objects, dynamic insertion, deletion, and renaming of methods, etc. Since HWOOPS inherits all the capabilities of Scheme, it is a very powerful object-oriented system.

The implementation of HWOOPS is aimed at fast message passing, somewhat at the cost of memory and object creation time. Despite the complexity of the created instance object—i.e., the superclass instance objects and their hierarchies—method values and superclass instance objects (explained below) are always found in linear time. This is regardless of where in the hierarchy the method is found. HWOOPS was primarily implemented to find the method value in constant time (using a hash table). However, it proved to be unnecessary for typical SCIX applications. It is very easy to change the evaluation mechanism in the macro to accomodate different search algorithms.

The following grammar describes a class definition:

```
class-definition  —→(define-class
                        (class-name class-par ...
                                   [&optional class-par ])
                        class-body )

class-body        —→[ (locals (loc-var loc-val) ...) ]
                    [ (inherit super ...) ]
                    [ (methods (method-name method-value) ...) ]
                    [ (init init-instr ...) ]

super             —→(class-name class-par ...)
                  |  parameter
```

A class object named `make-`*class-name* is defined. It accepts the number of *class-par* actual parameters. If the `&optional` directive is present the succeeding parameter will contain the list of the remaining actual parameters. The *class-body* consists of four parts: local variable bindings, inheritance of superclasses, method name bindings, and an initialization part.

The local variables have the same status as parameters. Their purpose is implement local state (i.e., binding of free variables) in the instance object. Inheritance is divided into two categories: the first category creates a new instance object of the specified class and inherits it, the second category inherits an already created instance object. In that way, sharing of superclass instance objects is implemented. Thus, the resulting object will be a Directed Acyclic Graph. The objects are the nodes and the vertices are oriented to the superclass objects. If you do not allow sharing, the general DAG will always be a tree structure. Methods are arbitrary values. The initialization part consists of an arbitrary number of Scheme instructions.

The scoping rules in *class-body* are simple: The parameters are visible throughout *class-body*. The local variables are visible commencing with the immediately succeeding local variable binding (if it exists) throughout *class-body*. The symbol *me* is visible in *class-body*. It always denotes the current object. So if you want to use the method $x$ defined in the same class, you only issue the

```
(me 'x ...)
```

command.

Message passing (i.e., the application of the instance objects) is described by the following grammar:

*message* $\longrightarrow$ *message-name* | (*class-name message*)

The symbol *message-name* denotes an element in the argument type of the instance object. The symbol *class-name* denotes a superclass object (since the object is structured by the use of inheritance, i.e., it is generated by several class object applications). The succeeding *message* is interpreted by that superclass instance object.

The value of the applied function object is returned. However, if this value is a procedure, it is automatically applied on the (possibly empty) rest of the argument list. The purpose for this is merely syntactical (eg., the command `(x 'me)` is more natural than the command `((x 'me))`; it gives a SMALLTALK-flavor to message passing). It also increases safety. If it is very easy to access procedure objects, one may be encouraged to do foolish things like storing them in other objects. If one really wants to access a procedure object it must be put in a thunk (a procedure of no arguments) illustrated by the following example:

```
(define-class (a-not-access b)
```

```
  (methods ; The value (lambda () b) is not accessible
   (b (lambda () b))))

(define-class (a-access b)
  (methods ; The value (lambda () b) is accessible
   (b (lambda () (lambda () b)))))

(define a-not (make-a-not-access 10))

(a-not 'b) ; Evaluates to the value b (i.e., 10)

(define a (make-a-access 10))

(a 'b) ; Evaluates to the procedure object

((a 'b)) ; Evaluates to the value b (i.e., 10)
```

The code in appendix C shows how a HWOOPS can be used for defining rather complex widgets.

## A.3 The Protocol

All instance objects created by HWOOPS support the following standard messages. If the resulting value is a procedure additional arguments, denoted $a$, $b$, ..., may be needed. Optional arguments are enclosed in square brackets.

`'supported-messages` Evaluates to a list of the supported messages of the object (excluding the standard messages).

`'object-desc` [$a$] Evaluates to a list containing the state of the object. The optional argument $a$ should be a thunk which evaluates to the calling object. The evaluated list is `exported` to the caller.

`'object-class`: Evaluates to *class-name*.

`'object-system`: Evaluates to `heavy-weight`, i e, describes the system.

`'insert-method!` $a$ $b$: Inserts the method `(`$a$ `.` $b$`)` in the object. Note, that since Scheme in general lacks first-class environments, the method value cannot refer to local state in the object. If this is an advantage or a disadvantage is debatable. A purist will probably call it an advantage.

`'delete-method!` *a*: Deletes the method named *a*.

`'rename-method!` *a b*: Renames the method *a* to *b*.

`'subclass-objects`: Evaluates to a list of the subclass instance objects.

`'superclass-objects`: Evaluates to a list of the superclass instance objects.

`'me`: Evaluates to the instance object itself.

The format of the list returned by the method `'object-desc` is as follows:

(*class-name method-list message-list internal-method-list superclass-object-list* )

The *method-list* element is an association list containing the methods to be exported. The *message-list* is a list of the exported messages. The *internal-method-list* is a list of the available internal methods. The *superclass-object-list* is a list of the superclass instance objects of the object.

By using the protocol defined above, it is very easy to use objects from other systems together with HWOOPS. Besides the protocol, HWOOPS requires the message `me` to be understood by all instance objects. Here follows an example:

```
(define-lw-class (a1 b)
  (methods
    ...
    (b (lambda () b))
    (set-b! (lambda (arg) (set! b arg)))
    ...
    ))

(define-class (a2 b)
  (inherit (a1 b)))

(define a (make-a2 10))
```

The instance objects generated by the class object `make-a1` must support the message `object-desc`. Something like:

```
(a1 ((b .  (lambda ...))  (set-b!  .  (lambda ...)))
    dont-care internal-method-list dont-care)
```

is returned.

Note the following subtle differences:

`(a 'b)` —→ Evaluates by the use of the exported *method-list* (of course, interned in `a`)

`(a '(a1 b))` —→ Evaluates by the use of the exported *internal-method-list*.

`((a '(a1 me)) 'b)` —→ Evaluates by the superclass instance object (this may be, and usually is, a completely different evaluation mechanism).

In general, it is by no means evident that all methods should be exported to the subclass instance objects. By using this subtle, but powerful, mechanism it is really possible to tune the system.

The `object-desc` should never be used by the user directly. It is even dangerous since it provides a mechanism to mutate the object.

## A.4 Examples

```
(x 'delete-method! 'event)
```

Return value is unspecified.

```
(let ((sym (gensym)))
  (x 'rename-method! 'event sym)
  (x 'insert-method! 'event (lambda (event)
                              (display event)
                              (x sym event) )))
```

Returns an unspecified value, but shows how easily you can redefine methods on the fly.

# B Portability

*After all, what good is a computer*
*if you can't take it with you?*
*— Tony Karp*

The Scheme code in *SCIX* follows [11] ($R^3RS$) as closely as possible. There are exceptions; some parts cannot be written using only this definition. For instance, an object-oriented system cannot be written without macros, a part of Scheme not formalized in $R^3RS$.

We used the `Scheme->C` system developed by Joel Bartlett at Digital Western Research Laboratory[6]. This Scheme compiles Scheme code into C code and links external C routines and compiled Scheme code into stand-alone applications, like new interpreters. With this system, it was easy to include the socket communication routines written in C.

## B.1 System Dependencies

To use the *SCIX* system together with another Scheme it would be necessary to either have some mechanism of including C routines in the Scheme or source code to the Scheme interpreter. It should be a simple task to port *SCIX* to one of the Scheme systems that are generally available with source code.

The object-oriented system is based on the `extend-syntax` macro expander described in [1].

As for hardware dependencies, the system requires the C types *short* and *short unsigned* to be 16 bits long, *unsigned* and *int* to be 32 bits, and *char* to be 8 bits. These dependencies all lie in the conversion routines, written in C, therefore easy to fix.

Porting the system to a non-Unix system would require rewriting the low-level C routines used for server communication. Again, this should not pose any problems.

## B.2 Non-Standard Functions and Syntax Used

The *SCIX* system uses some functions and syntax not specified as "essential" in $R^3RS$.

### Syntax and procedures appearing in $R^3$RS

Syntax:

`if`-statements without <*alternate*> [11, page 8]

`case` [11, page 8]

`and` [11, page 9]

`or` [11, page 9]

`let*` [11, page 9]

"Named let" [11, page 10]

`` ` `` and `,` [11, page 11]

All `define`-forms [11, page 11]

Procedures:

`append` with variable number of operands [11, page 16]

`list-tail` [11, page 16]

`list-ref` [11, page 16]

`last-pair` [11, page 16]

`modulo` [11, page 20]

`round` [11, page 20]

`inexact->exact` [11, page 21]

`make-string` [11, page 24]

`string-set!` [11, page 24]

`map` with variable number of operands [11, page 26]

(Some of these have changed status in [12])

### Syntax and procedures not appearing in $R^3$RS

Syntax:

`extend-syntax` [1, page 175ff]

Procedures:

`remq` Remove element with equivalence predicate `eq?`

`remq!` As above, but destructive.

`remove` As `remq`, but equivalence predicate `equal?`

`bit-and` Bitwise and.

`bit-or` Bitwise or.

`bit-lsh` Left shift.

`bit-rsh` Right shift.

`gensym` Returns a symbol not equal (in any sense) to any other symbol.

`string-append!` Copies a string into an existing buffer

`c-`<*type*>`-ref` Reference a <*type*> from a C-pointer

`c-byte-set!` Set value of a byte referenced by a C-pointer.

# C An Extended Example

*If you cannot convince them, confuse them.*
*— Harry S Truman*

The code for a button widget:

```
;; Author: Magnus Persson <mpersson@stkhlm.dec.com>

;; A generic BUTTON is defined below. A button consists of two borders
;; (an outer border which is always highlighted and an inner border which
;; is highlighted when the button is pressed). Within the inner border an
;; area is located.

;; The class accepts the following parameters:
;;    WIDTH -- The width of the button.
;;    HEIGHT -- The height of the button.
;;    X, Y   -- The coordinates of button in the PARENT window.
;;    AREA -- The area within the inner borders.
;;    BUTTON-SEMANTICS -- A procedure object of one argument.
;;                        The action to take when a pressed button
;;                        is released (arg. ApplyApplication) or
;;                        the button is reset (arg. Reset)
;;    GC-DRAW, GC-REV, GC-INV -- GC's to draw (highlight), reverse (not
;;                               highlight), and invert.
;;                               The outer border uses GC-DRAW.
;;                               The inner border uses GC-DRAW and GC-REV.
;;                               The area uses GC-INV. None of these
;;                               may be #F.
;;    PARENT -- The window where to put the button.
;;    SCR -- The screen.
;;
;; The class inherits:
;;    WINDOW -- The button window.
;;    DFSM -- Using a specification of a generic button.
;;
;; The instance objects accepts the following messages:
;;    RESET reset-log (optional) -- Reset the button. Reset-log decides
;;                                  if to log the button (convenient for
;;                                  replaying the actions of a button).
;;    TRANSIT input -- Refine the method found in the DFSM. All the
;;                     flushing is done here (except when the button
;;                     is reset).
;;    ACTIVATE -- To activate the button; i.e., create and map its window.
```

```
;;    DESTROY -- To destroy the window.
;;    SCIX-EVENT-DISPATCH ev-source obj -- Used for event handling.

;;; Generic button.
;;;
(eval-when (compile load eval)
  (define-class (button width height x y area button-semantics
                       gc-draw gc-rev gc-inv parent scr)
    (locals
     (i #f)    ;; For caching
     (value-mask (make-window-value-mask
                 `(event-mask ,(make-event-mask 'ButtonPress
                                                'ButtonRelease
                                                'EnterWindow
                                                'LeaveWindow
                                                'Exposure))))
     (n-border (list (make-polyrectangle
                     `((0 0 ,(- width 1) ,(- height 1))))))
     (w-border (list (make-polyrectangle
                     `((1 1 ,(- width 3) ,(- height 3)))))) )
    (inherit
     (window width height (parent 'depth) x y parent 0
            'CopyFromParent 'CopyFromParent scr #f)
     (dfsm
      `(((EnterNotify . DisArmed) Armed
                                 ,(lambda () (me 'draw w-border gc-draw)))
        ((LeaveNotify . DisArmed) DisArmed)
        ((LeaveNotify . Armed) DisArmed
                              ,(lambda () (me 'draw w-border gc-rev)))
        ((LeaveNotify . Pressed) DisArmed
                                ,(lambda ()
                                   (me 'draw w-border gc-rev)
                                   (me 'draw area gc-inv)))
        ((ButtonPress . Armed) Pressed
                              ,(lambda ()
                                 (me 'draw area gc-inv)))
        ((ButtonRelease . Armed) Armed)
        ((ButtonRelease . DisArmed) DisArmed)
        ((ButtonRelease . Pressed) Armed
                                  ,(lambda ()
                                     (button-semantics
                                       'ApplyApplication))) )
      'Disarmed #t))
    (methods
     (reset (lambda reset-log
             (me '(dfsm reset) (lambda ()
                                 (me 'draw n-border gc-draw)
```

```
                                  (me 'draw w-border gc-rev)
                                  (button-semantics 'Reset))
                  (not (null? reset-log)))
              (scr 'flush!)))
     (transit (lambda (input)
                (me '(dfsm transit) input)
                (scr 'flush!)))
     (activate (lambda ()
                 (me 'createwindow value-mask)
                 (me 'mapwindow)
                 (me 'reset)))
     (destroy (lambda ()
                (me 'destroywindow) ))
     (scix-event-dispatch (lambda (ev-source obj)
                            (let* ((ev-l (if (pair? ev-source)
                                             ev-source
                                             (ev-source 'my-events! obj))))
                              (let loop ((ev-l ev-l))
                                (if (not (null? ev-l))
                                    (let* ((e (car ev-l))
                                           (event-name (e 'event-name)) )
                                      (cond ((eq? event-name 'Expose)
                                             (if (zero? (e 'count))
                                                 (me 'reset))
                                             (loop (cdr ev-l)))
                                            ((memq event-name i)
                                             (me 'transit event-name)
                                             (loop (cdr ev-l)))
                                            (else
                                             (format #t "B got ~a~%"
                                                     event-name))))))))))
    (init
     (set! i (me 'i))     ;; Cache input alphabet
     (me 'scix-announce-id! me)))
  ) ;; End of eval-when

;; A PUSH BUTTON is defined below. A push button has an application
;; associated with the release of the pressed button.

;; The class accepts the following parameters:
;;    X, Y   -- The coordinates of the button in the PARENT window.
;;    CONTENTS -- An area structure (see above) of arbitrary complexity -
;;                it could be some recursive structure using a combination
;;                of filled areas, texts, and pixmaps. The only additional
;;                cost for drawing complex figures occursp when the push
;;                button is reset!
;;    GC-DRAW, GC-REV, GC-INV -- GC's to draw (highlight), reverse (not
```

```
;;                                highlight), and invert.
;;                                The outer border uses GC-DRAW.
;;                                The inner border uses GC-DRAW and GC-REV.
;;                                The area uses GC-INV. None of these may
;;                                be #F.
;;    GC-AREA, GC-CONT -- GC's when the button is redrawn. These may be #F.
;;                         In that case CONTENTS contain bound GC's for
;;                         drawing.
;;    PARENT -- The window where to put the button.
;;    APPLICATION -- The associated application (must be a thunk, i.e.,
;;                    a procedure of no arguments).
;;    SCR -- The screen.
;;
;; The class inherits:
;;    BUTTON -- The generic button.

;;; Push-button
;;;
(eval-when (compile load eval)
  (define-class (push-button x y contents gc-draw gc-rev gc-inv
                             gc-area gc-cont parent application scr)
    (locals
     (area (contents 'area))
     )
    (inherit
     (button (+ 4 (contents 'width)) (+ 4 (contents 'height))
             x y area (lambda (input)
                        (case input
                          ('Reset (contents 'draw me gc-area gc-cont))
                          ('ApplyApplication
                           (me 'draw area gc-inv)
                           (application))))
             gc-draw gc-rev gc-inv parent scr)
     ))
  )

;; A TOGGLE BUTTON is defined below. A toggle button has two applications
;; associated with it. One application is used when the button is toggled
;; to be set and one application is used when the button is toggled to be
;; reset.

;; The class accepts the following parameters:
;;    X, Y   -- The coordinates of button in the PARENT window.
;;    CONTENTS -- An area structure (see above) of arbitrary complexity -
;;                it could be some recursive structure using a combination
;;                of filled areas, texts, and pixmaps. The only additional
;;                cost for drawing complex figures occurs when the toggle
```

```
;;                button is redrawn!
;;    GC-DRAW, GC-REV, GC-INV -- GC's to draw (highlight), reverse (not
;;                               highlight), and invert.
;;                               The outer border uses GC-DRAW.
;;                               The inner border uses GC-DRAW and GC-REV.
;;                               The area uses GC-INV. None of these may
;;                               be #F.
;;    GC-AREA, GC-CONT -- GC's when the button is redrawn. These may be #F.
;;                        In that case CONTENTS contain bound GC's for
;;                        drawing.
;;    PARENT -- The window where to put the button.
;;    APPLICATION-ON -- The application when toggled to be set (a thunk).
;;    APPLICATION-OFF -- The application when toggled to be reset (a thunk).
;;    SCR -- The screen.
;;
;; The class inherits:
;;    BUTTON -- The generic button.
;;    DFSM -- Specifies the toggle states and transitions. An alternative
;;            is to augment the DFSM in BUTTON (not good) or even to
;;            rewrite it (even worse). In this way, the number of total
;;            states is reduced.

;;; Toggle-button.
;;;
(eval-when (compile load eval)
  (define-class (toggle-button x y contents gc-draw gc-rev gc-inv
                               gc-area gc-cont parent application-on
                               application-off scr)
    (locals
     (area (contents 'area))
     )
    (inherit
     (button (+ 4 (contents 'width)) (+ 4 (contents 'height)) x y area
             (lambda (input) (me '(dfsm transit) input))
             gc-draw gc-rev gc-inv parent scr)
     (dfsm `(((ApplyApplication . R) S ,application-on)
             ((ApplyApplication . S) R ,application-off)
             ((Reset . Undefined) R
                                  ,(lambda ()
                                     (contents 'draw me gc-area gc-cont)))
             ((Reset . R) R  ,(lambda ()
                                (contents 'draw me gc-area gc-cont)))
             ((Reset . S) S  ,(lambda ()
                                (contents 'draw me gc-area gc-cont)
                                (me 'draw area gc-inv))))
           'R #f) ))
  ) ;; End of eval-when
```

The `dfsm` that appears in the above is a deterministic finite state machine constructor. Albeit elegant, it is somewhat periferal to this report. The important thing to note is that the various buttons inherit the dfsm as well as the window class.

## References


[1] R. Kent Dybvig. *The Scheme Programming Language*. Prentice-Hall, New Jersey, 1987.

[2] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, 1985.

[3] James Gettys, Robert W. Scheifler, and Ron Newman. *The X Window System: C Library and Protocol Reference*. Digital Press, Burlington MA, 1988.

[4] David Rosenthal James Gosling and Michelle Arden. *The NeWS Book*. Springer-Verlag, New York, 1989.

[5] *CLX*. Texas Instruments, Inc., P.O. Box 2909, Austin, 1987.

[6] Joel Bartlett. *Scheme->C, a Portable Scheme-to-C Compiler*. Technical Report 89/1, Digital Western Research Laboratory, Palo Alto, January 1989.

[7] Joel McCormack, Paul Asente, and Ralph R. Swick. *X Toolkit Intrinsics — C Language Interface*. M.I.T., Project Athena, 1988.

[8] Douglas A. Young. *X Window Systems Programming and Applications with Xt*. Prentice-Hall, New Jersey, 1989.

[9] David S. H. Rosenthal. *Inter-Client Communication Conventions Manual*. 1989.

[10] Oliver Jones. *Introduction to the X Window System*. Prentice-Hall, New Jersey, 1989.

[11] Jonathan Rees and William Clinger, editors. The Revised$^3$ Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices*, December 1986.

[12] William Clinger and Jonathan Rees, editors. The Revised$^{3.99}$ Report on the Algorithmic Language Scheme. August 1989. Draft obtainable from zurich.ai.mit.edu.