MysterX: A Scheme Toolkit for Building Interactive Applications with COM

Paul A. Steckler*
Department of Computer Science
Rice University, MS 132
6100 S. Main St.
Houston, TX 77005-1892
steck@cs.rice.edu

Abstract

MysterX is an object-oriented Scheme toolkit for building applications from off-the-shelf COM components. While the COM support in languages such as Haskell and Mercury requires the use of an interface compiler to generate stub code, MysterX uses the reflective capabilities of OLE Automation to make value-marshalling decisions at run-time. MysterX hosts COM components in windows that display Dynamic HTML, without requiring a separate browser. Scheme code can manipulate HTML elements and their style properties in such windows to create interesting visual effects. Event handlers written in Scheme can be associated with HTML elements and COM objects. By integrating these diverse technologies, MysterX can be used to write complete GUI applications.

1: Introduction

MysterX is an object-oriented Scheme toolkit for building interactive applications with components that adhere to the Component Object Model (COM). COM allows applications to be built from off-the-shelf components. Components and their host programs can be written in any language; a component and its host program may be written in different languages. Most often, COM components are contained in applications built with Microsoft's Visual Basic or with various implementations of C++. Another way of using COM components is to "script" them in Web pages containing code in either JavaScript or VBScript, a browser-hosted version of Visual Basic. Unfortunately, these usual approaches of handling COM components rely on languages that make the task somewhat cumbersome. To overcome this problem, several research groups have developed COM support for languages with clean syntax, structure, and semantics. MysterX, Rice's effort at adding COM support for Scheme, uses Scheme to particular advantage.

By combining other technologies with COM, MysterX is able to use COM components in dynamic, interactive applications. A COM component instance is an object with state; most components have some visual representation. Displayable COM objects are ordinarily hosted in a container, such as a Web browser. In MysterX, COM components are hosted in a window containing Dynamic HTML, though no Web browser is run directly. Therefore,

* The author was partially supported by NSF grant CDA-9713032 for this work.

applications created with MysterX can use any HTML elements, such as bitmaps, buttons, drop-down lists, frames, and tables, as well as text. In MysterX programs, Scheme code handles Dynamic HTML events, and can be used to modify HTML elements, in the same way that code written in a client-side Web scripting language, such as JavaScript, performs these tasks in a browser. MysterX extends the capabilities of Dynamic HTML, a Web-based technology, to conventional applications. MysterX thus adds more to Scheme than just COM support — it adds the ability to build full-featured GUI programs.

The remainder of the paper is organized as follows. In Section 2, we describe the document abstraction provided by MysterX; in Section 3, we show how the HTML elements in a document can be modified; in Section 4, we explain how MysterX programs use the reflective capabilities of OLE Automation to get descriptions of methods and properties, and how methods and properties are used; in Section 5, we explain how events are generated in applications, and how they can be handled by Scheme code; Section 6 describes the technical details of the MysterX implementation; we present conclusions in Section 7.

2: MysterX documents

MzScheme, Rice's Scheme interpreter, supports classes for object-oriented programming [Fla98]. Each MysterX window contains a *document*, which is an instance of the Scheme class `mx-document%`. A MysterX document roughly corresponds to the notion of document in the Document Object Model, which is the official basis for Dynamic HTML [Con99]. A MysterX program may create arbitrarily many documents. HTML elements and COM objects in distinct documents may communicate with one another via Scheme glue code.

To create an instance of `mx-document%`, the syntax is:

(`make-object mx-document%` *label width height x y style*)

where each of the arguments is optional. The *style* argument is a list of options for the resulting window, such as its border and menu. An `mx-document%` instance is a first-class Scheme value. A new instance of a document appears as an empty window.

There are just a few public methods of `mx-document%`. Those that deal with COM objects are:

- `objects`: returns a list of the COM objects, including ActiveX controls, contained in the document. Each object has the Scheme type `<com-object>`, which encapsulates the COM `IDispatch` interface.
- `insert-object` *coclass*: places an instance of the COM class *coclass* at the beginning of the document.
- `append-object` *coclass*: like `insert-object`, but places the instance at the end of the document.

Note that the *coclass* argument of `insert-object` and `append-object` is a string naming the coclass. Such a string is not guaranteed to uniquely name a COM class, but probably will on a given user's machine. COM classes are uniquely named by a CLSID, a 128-bit descriptor, but these are inconvenient to use. If there is any concern about ambiguity, a MysterX programmer can use a CLSID directly in HTML.

The HTML-related public methods of `mx-document%` are:

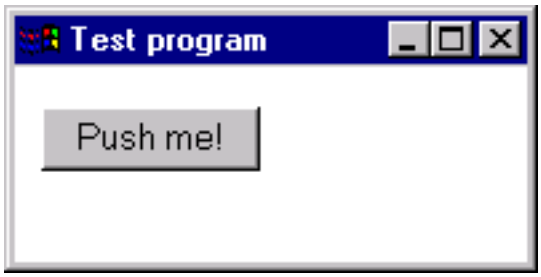

Figure 1. A MysterX document.

- `insert-html` *s*: inserts its string argument *s* as HTML at the beginning of the document.
- `append-html` *s*: like `insert-html`, but places the HTML in *s* at the end of the document.
- `replace-html` *s*: replaces all the HTML in the document by the HTML in *s*.
- `find-element` *tag id*: returns the HTML element with the given HTML tag and "id" attribute, as an instance of the Scheme class `mx-element%`.

In order for `find-element` to work as indicated, programmers must be sure to include the "id" attribute in the start tag of HTML elements that they wish to manipulate from Scheme.

Here is an example of how these methods are used. The following Scheme code creates a document, inserts a pushbutton, and binds a variable to the pushbutton:

```
(define document (make-object mx-document% "Test program"))
(send document insert-html "<BUTTON id=\"my-button\">Push me!</BUTTON>")
(define button (send document find-element "BUTTON" "my-button"))
```

The resulting window is shown in Figure 1.

HTML elements generate *events*, which can be handled from Scheme. HTML events are instances of the Scheme class `mx-event%`, which we describe in Section 5. The `mx-document%` methods relating to events are

- `register-event-handler` *elt fn*: registers an event handler *fn* for the HTML element *elt*. The handler *fn* is a Scheme procedure of one argument, which will be an event.
- `unregister-event-handler` *elt*: removes any existing event handler for the HTML element *elt*.
- `handle-events`: spawns a Scheme thread for handling events for the elements in the document.
- `stop-handling-events`: kills any existing event handler thread for the elements in the document.

We provide more details of event-handling, including event-handling for COM objects, in Section 5.

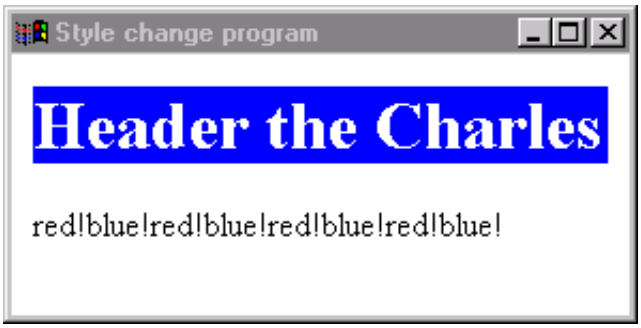

Figure 2. Changing the style of an HTML element.

3: Modifying HTML elements

Most HTML elements have a *style* attribute, which indicates visual properties such as width, height, background color, font size, and so forth. The style attribute can be supplied in the start tag for an element, or style properties can be assigned to HTML tags in special `STYLE` blocks. There are complex rules for the inheritance of styles by elements from other elements. For a given element, its style attribute may be derived from several sources. The "Cascading Style Sheet" specification, part of the Document Object Model, indicates how styles flow from element to element [Con99].

MysterX applications can modify the style attribute properties of HTML elements. There are more than eighty such properties. With a few exceptions, the `mx-element%` class provides a pair of "get" and "set" methods for each style attribute property. As an example, suppose we wanted to create a page with a header whose background color changes when the mouse position changes. Assuming `document` is a variable bound to a MysterX document, we could write:

```
(send document insert-html
      (string-append "<H1 id=\"my-header\""
                     "style=\"color: white background-color: blue\">"
                     "Header the Charles</H1>"
                     "<p>"))

(let ([elt (send document find-element "H1" "my-header")])
  (send document register-event-handler elt
          (lambda (event)
            (cond
              [(send event mouseover?)
               (send elt set-background-color! "red")
               (send document append-html "red!")]
              [(send event mouseout?)
               (send elt set-background-color! "blue")
               (send document append-html "blue!")]))))

(send document handle-events)
```

The `<H1>` header background color is initially blue, but the color changes as the mouse is

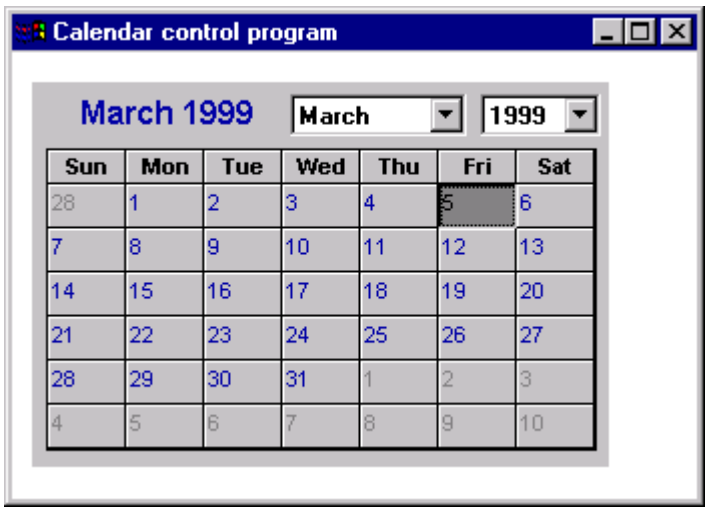

Figure 3. Calendar control, a COM object.

moved over or off the text. With each color change, the name of the new color is appended to the document. Figure 2 shows a document after several such changes.

MysterX also provides a "raw" interface to style properties via the method `set-css-text!`. Programmers can use HTML syntax for those properties. In the program fragment above, to set the background color to red, we could have written

```
(send elt set-css-text! "color:red")
```

An advantage of the raw interface is that many properties can be set at once:

```
(send elt set-css-text! "color:red font:sans-serif font-size:24pt" )
```

Besides allowing the modification of style properties of HTML elements, MysterX allows HTML to be modified by insertion of new elements and by the replacement of existing elements. The `insert-html` and `append-html` methods for `mx-document%`, mentioned earlier, allow placement of new HTML only at the beginning and end of the entire document; the same-named methods for `mx-element%` allow placement of new HTML before and after individual elements. Likewise, the `replace-html` method for `mx-document%` replaces the entire contents of the document; the same method for `mx-element%` replaces just the element.

4: Methods, properties, and COM reflection in Scheme

COM objects, including ActiveX controls, can be inserted into MysterX documents using the `insert-object` and `append-object` methods of `mx-document%`. Alternatively, given a COM class name, the Scheme procedure `coclass->html` returns an HTML string which can be used to load a COM object. For example,

```
(define calendar (send document insert-object "Calendar Control 8.0"))
```

binds `calendar` to an instance of the COM class "Calendar Control 8.0", which appears as in Figure 3. Once we have a reference to a ⟨`com-object`⟩ value, we can invoke its methods and use its properties. But how do we know what those methods and properties are, and

how to use them?

MysterX has the procedure `com-help`, which takes a ⟨`com-object`⟩ argument and an optional topic name. This procedure starts either the WinHelp system or the HTML Help system, as appropriate, with the documentation for the COM class that generated the object. Such documentation is often useful in determining how the methods and properties of an object are meant to be used. WinHelp and HTML Help files for COM components are usually pitched to Visual Basic programmers, so the language syntax in these files may be unenlightening to Scheme programmers.

Fortunately, most COM objects that support OLE Automation offer a reflective capability through the `ITypeInfo` interface. With this interface, a programmer can query an object to determine the names of its methods and properties. For example:

```
> (com-methods calendar)
("AboutBox" "Today" "Refresh" "PreviousYear" "PreviousWeek"
 "PreviousMonth" "PreviousDay" "NextYear" "NextWeek" "NextMonth"
 "NextDay")
```

Similarly, we can obtain `calendar`'s readable properties with

```
> (com-get-properties calendar)
("Year" "ValueIsNull" "_Value" "Value" "TitleFontColor"
 "TitleFont" "ShowVerticalGrid" "ShowTitle"
 "ShowHorizontalGrid" "ShowDays" "ShowDateSelectors"
 "MonthLength" "Month" "GridLinesColor" "GridFontColor"
 "GridFont" "GridCellEffect" "FirstDay" "DayLength"
 "DayFontColor" "DayFont" "Day" "BackColor")
```

Likewise, the procedure `com-set-properties` returns a list of writable properties for its ⟨`com-object`⟩ argument.

To use a method or property correctly, we need to know its type. The `ITypeInfo` interface can also provide such information for most OLE Automation objects. MysterX primitives access the information provided by `ITypeInfo`. For example, we have

```
> (com-method-type calendar "PreviousWeek")
(-> void)
```

meaning the method takes no arguments, and the return value is the special MzScheme `#<void>` value. This function type is meaningful to Scheme programmers, and is the translation of the type provided by COM. It turns out that the types of all the `calendar` methods above are the same — hence all these methods are used for their side effects.

The Calendar Control properties have slightly more interesting types:

```
> (get-property-type calendar "Day")
(-> short-int)
```

Therefore, this property is called with no arguments, and returns a 2-byte integer. The type `short-int` is not a native Scheme type, of course. In return position, distinguishing this type from the ordinary Scheme integer type is usually unimportant, but in argument position, that distinction is critical: passing a Scheme bignum is a type violation. MysterX detects such type violations when methods or properties are invoked.

Curiously, we have:

```
> (get-property-type calendar "Value")
(-> mx-any)
```

This type indicates that the return value might have any type producible by a COM object. The `mx-any` type is a translation of the COM type `VARIANT`. When it appears in argument position, `mx-any` means we can pass a method or property *any* Scheme value that has a meaningful COM equivalent, and the COM object will try to use that value.

With types in hand, we can experiment with COM methods and properties. Method invocation uses the procedure `com-invoke`, which takes a COM object, the name of a method, and any arguments. Property reading and writing use the procedures `com-get-property` and `com-set-property!`, respectively. For the readable properties mentioned above, we have

```
(com-get-property calendar "Day")
```

which returns an integer, and

```
(com-get-property calendar "Value")
```

returns a value of type ⟨`com-date`⟩, which is the Scheme version of the COM type `DATE`.

When invoking methods and using properties, Scheme argument values are checked to make sure they are compatible with the COM type given by the `ITypeInfo` interface. If so, the values are marshalled to COM equivalents. Return values are not checked, since we can rely on the safety properties offered by the COM object. OLE Automation return values always come packaged as a VARIANT type, even if the advertised return type is more specific. MysterX unmarshals these values to an appropriate Scheme type.

We wish to highlight the difference between our approach to COM and that used by COM research projects for other languages. The Haskell community has described their work with COM in a series of papers [FLMJ98, JML98, MLH99]. Haskell uses the type descriptions in a COM interface description language (IDL) file, and uses a compiler to generate stub code for gluing with C++. Pucella has built an IDL-to-Standard ML compiler, which can be used to support COM [Puc98]. Similarly, Mercury, a mixed logic and functional language, also compiles IDL specifications to generate bindings for CORBA [JDS99].

These implementations gain the benefit of strong typing in the languages involved, at the cost of a preprocessing step. In contrast, we avoid a preprocessing step, and take advantage of the reflective capabilities of OLE Automation objects. This approach takes the dynamic typing and interactivity of Scheme as its strength. We do lose the ability to use those relatively few COM classes that do not support `IDispatch`, the Automation interface. We also incur a small performance penalty in checking the types of arguments for each COM method invocation, though such a penalty is usual in Scheme. It is possible to add support for non-Automation classes by adding a preprocessing step, though we have not yet felt compelled to do so.

5: Events in MysterX

Events can be generated either by HTML elements or by COM objects. The mechanisms for handling events from these two sources are distinct.

As we have seen, the `mx-document%` class has methods for registering event handlers associated with HTML elements. HTML events are instances of the Scheme class `mx-event%`.

An `mx-event%` instance encapsulates the COM interface `IHTMLEventObj`. An event handler can identify the kind of event that occurred by using predicate methods such as `keypress?`, `keydown?`, and `mousedown?`. The `mx-event%` class also has methods that describe the event in various ways. For instance, the `mousebutton` method returns an integer indicating which mouse button was pressed.

Since MysterX uses the rendering engine underlying Internet Explorer 4, events are propagated as in that browser. Since events bubble up the tree of HTML elements, MysterX listens for events only at the root, the `<BODY>` of the HTML page, not at individual elements. Since events carry tag and attribute information, the MysterX event dispatcher can readily determine which element generated a particular event.

While there is a fixed set of HTML events, COM objects can generate both "standard" events, defined by the ActiveX controls specification, or "custom" events, defined by a particular COM class [Den97]. MysterX uses COM reflection to obtain the events supported by a COM object, and the types required for each event's handler. For the calendar control we saw in the last section, we have

```
> (com-events calendar)
("NewYear" "NewMonth" "AfterUpdate" "BeforeUpdate" "KeyUp"
 "KeyPress" "KeyDown" "DblClick" "Click")
```

and

```
> (com-event-type calendar "Click")
(-> void)
```

We can install a handler for the `Click` event with

```
(com-register-event-handler calendar "Click"
                            (lambda () (printf "Got a click!")))
```

6: Implementation

MysterX relies on two key technologies in its implementation. The first is the functionality offered by the Microsoft Active Template Library (ATL) to load COM objects into a window; the other is MzScheme's extension feature. The implementation consists of three dynamic libraries (.DLL's), MysPage, MysSink, and MysterX, and a small Scheme library.

The MysPage .DLL implements a Dynamic HTML COM class, which encapsulates an HTML page. A instance of this COM class is created for each MysterX document. ATL provides the functionality to host the Dynamic HTML COM object in a window. The underlying HMTL page for the COM object is blank, except for a `<BODY>` containing event handler declarations such as

```
onmousedown='window.external.AtAnyEvent()'
```

and so on for each event we wish to trap. These handler declarations are not visible when the HTML page is rendered. `AtAnyEvent()` is a compiled C++ procedure that is called for each event in the document. The DHTML class has an associated event queue, which is also implemented as a COM class.

The MysSink .DLL implements a COM event sink object that dispatches COM events to registered Scheme handlers.

The MysterX .DLL contains compiled C++ code which implements various primitives callable from Scheme. We use MzScheme's `load-extension` facility to load the .DLL, extending the current environment with bindings for the MysterX primitives. Each new document creates a new window with its own Windows message-handling loop. That loop needs to run in a Win32 thread separate from the MzScheme read-eval-print (REPL) loop, so the latter does not get blocked. As mentioned, `<com-object>` values encapsulate `IDispatch` interface pointers; such values are available from the MzScheme REPL. Unfortunately, COM interface pointers are valid only in their creating thread. To obtain interface pointers that are valid in the REPL, we use the inter-thread interface marshalling features of Distributed COM (DCOM). In fact, we only need to use these features explicitly when obtaining interfaces for the DHTML object and its associated event queue object. At that point, we create proxy versions of those objects, which we can deal with directly in the REPL thread. Thereafter, whenever we obtain a new COM interface, DCOM automatically and invisibly creates a proxy.

We briefly mention an abandoned implementation strategy, which relied on DCOM even more heavily. In that version, a Dynamic HTML COM object was hosted in the Internet Explorer 4 browser, instead of windows directly associated with MzScheme. Therefore, COM objects resided in a separate *process*, and we used DCOM to handle all interprocess communication. Ultimately, we rejected the use of the browser as too cumbersome for users, and the DCOM implementation as too complex.

7: Conclusions

We have built an object-oriented toolkit for GUI applications in Scheme. Our older GUI toolkit for MzScheme, MrEd [FF98], is full-featured, but MysterX can do things that MrEd cannot. MysterX's use of OLE Automation fits in well with interpretive nature of Scheme. A MysterX programmer can try out the methods and properties of an ActiveX control from the MzScheme REPL prompt. By wrapping documents, HTML elements, and events in OO classes, MysterX provides a convenient programming model, much like the one successfully used in MrEd. In fact, the two toolkits can be combined, if desired, to get the capabilities of both.

When we began work on MysterX, we primarily sought the ability to use ActiveX controls and other COM objects. We have achieved that, though not quite in the way we had envisioned. In particular, we did not contemplate using Dynamic HTML for hosting COM objects. Once we went down that path, it became clear that having DHTML available in applications was useful. It is apparent that DHTML is valuable not only for Web pages, but also for many conventional applications.

Let us mention some advanced features offered by MysterX via Dynamic HTML that are not easy to emulate in MrEd. One such feature is the ability to embed Java applets by using the `APPLET` tag, providing another source of software components. Another useful MysterX feature is the ability to use visual filters. Filters transform the appearance of HTML elements. For example, there are filters for grayscaling, flipping an image horizontally and vertically, and blurring. Figure 4 shows a button that has been flipped horizontally. There is much functionality in the Internet Explorer 4 rendering engine that MysterX can use for its own ends.

We would like to test MysterX with a wider variety of COM components, and build

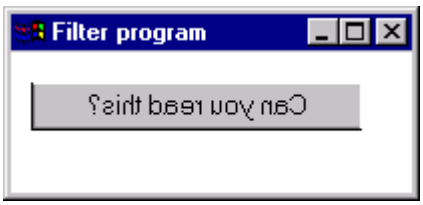

Figure 4. Applying a visual filter to an HTML element.

some large applications. Possible future enhancements include support for non-Automation components and tighter integration into the MrEd framework.

Acknowledgements

The author thanks John Stone and members of Rice PLT for useful comments on this paper. Jim Bullard bravely tested various iterations of MysterX. Microsoft supplied software and documentation.

References

[Con99] World Wide Web Consortium. Document Object Model (DOM) Level 2 Specification. `http://www.w3.org/TR/WD-DOM-Level-2`, Mar. 1999.

[Den97] Adam Denning. *ActiveX Controls Inside Out*. Microsoft Press, 2d. edition, 1997.

[FF98] Matthew Flatt and Robert B. Findler. PLT MrEd: Graphical toolbox manual. Part of PLT software distribution, July 1998. Version 53.

[Fla98] Matthew Flatt. PLT MzScheme: Language Manual. Part of PLT software distribution, July 1998. Version 53.

[FLMJ98] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. *H/Direct*: A binary foreign language interface for Haskell. In *International Conference on Functional Programming (ICFP'98)*, pages 153–162, New York, 1998. ACM.

[JDS99] David Jeffery, Tyson Dowd, and Zoltan Somogyi. MCORBA: A CORBA binding for Mercury. In Goyal Gupta, editor, *Practical Aspects of Declarative Languages (PADL'99)*, number 1551 in Lecture Notes in Computer Science, pages 211–227, Berlin, 1999. Springer-Verlag.

[JML98] Simon Peyton Jones, Erik Meijer, and Daan Leijen. Scripting COM components in Haskell. In *Proc. Fifth International Conference of Software Reuse*, page unknown, 1998.

[MLH99] Erik Meijer, Daan Leijen, and James Hook. Client-side web scripting with HaskellScript. In Goyal Gupta, editor, *Practical Aspects of Declarative Languages (PADL'99)*, number 1551 in Lecture Notes in Computer Science, pages 196–210, Berlin, 1999. Springer-Verlag.

[Puc98] Riccardo Pucella. An overview of the ML-IDL compiler (Technical Note #1). `http://cm.bell-labs.com/who/riccardo/notes/overview-ml-idl`, Nov. 1998.