Component Support in PLT Scheme

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

Abstract

PLT Scheme (DrScheme and MzScheme) supports COM components with two pieces of software. The first piece is MzCOM, a COM class that makes a Scheme evaluator available to COM clients. With MzCOM, programmers can embed Scheme code in programs written in mainstream languages such as C++ or Visual BASIC. Some applications can also load MzCOM. The other piece of component-support software is MysterX, which makes COM classes available to PLT Scheme. When needed, MysterX uses a programmable Web browser to display COM objects.

1 Introduction

The Component Object Model (COM) opens a wide channel for Scheme programs to interact with "real-world" software. On one end of that channel, many commercial applications on the Windows platform have representations as COM classes that could profitably be used by Scheme programs. For example, Internet Explorer and most of the applications in Microsoft's Office suite can be scripted via COM. Programmers working with mainstream languages such as C++, Visual BASIC, Javascript, and Delphi have been able to script those applications for several years. Recently, there has been considerable activity in the functional language community in using COM for component scripting, demonstrating the eminent practical benefits of functional languages [JML98, Ler99, Ste99]. On the opposite end of the channel, software written in mainstream languages can benefit from the facilities offered by Scheme, if made available from a COM class. A C++ program might wish to perform computations that require Scheme's bignums, for instance, or use a library offered by the Scheme implementation. Some applications also allow direct connections to COM classes via their scripting languages, so Scheme can benefit them as well.

PLT has created software placed at both ends of the COM channel. One piece of software, MzCOM, makes a PLT Scheme evaluator available as a COM class. The other piece of software, MysterX, makes COM classes available to PLT Scheme. MysterX has additional features, such as a programmable integrated Web browser, that make it useful beyond COM.

In this paper, we present MzCOM and MysterX, and how they are used. In Section 2, we describe MzCOM and how to use it with various programming languages. In Section 3, we describe MysterX and how it may be used to script COM components.

2 MzCOM

MzCOM is a COM class wrapped around the MzScheme support libraries. The code itself is written in C++. With MzCOM, any COM client can run Scheme code.[1] It is compiled as a dual-mode class, so that its methods can be called directly or interpretively, using OLE Automation.[2]

We have designed MzCOM as an *out-of-process* COM server, which offers several advantages. First, a failure in MzCOM will not bring down its client. Second, a client can instantiate any number of MzCOM instances. The MzScheme libraries used to implement MzCOM are non-reentrant. Hence, multiple instances of an in-process server would yield at best undesired sharing of state, and at worst, a crash. Third, MzCOM can be easily "remoted", or run across a network connection.[3]

A COM client accesses MzCOM by obtaining its `IMzObj` interface. This interface contains three methods

```
 - Eval : BSTR x BSTR * -> HRESULT
 - Reset : -> HRESULT
 - About : -> HRESULT
```

where the types are those one would use in C or C++. `BSTR` is the COM type for strings ("BASIC string"). `HRESULT` is a type used by COM methods to return error codes. The `About` method pops up an informational dialog. `Reset` restores a pristine Scheme environment. `Eval` evaluates a sequence of S-expressions formatted into a string. Bindings are retained between subsequent calls to `Eval`, allowing us to define procedures in one call, then apply them later. The result of Scheme evaluation is returned through the second argument to `Eval`, a pointer to a string.

Returning a string as the evaluation result is a design compromise. COM allows a limited set of types for the return types of COM methods. While the inhabi-

[1] A similar effort, ScriptServer, allows Haskell and other languages to be used in the limited context of COM scripting [Lei99]

[2] See Chapter 14 of [Szy98] for an introduction to COM and related technologies. [Box98] describes dual-mode classes and other technical aspects of COM.

[3] Using recent versions of COM, in-process servers can be remoted using a "surrogate process" on the server machine, with some complicated setup.

```
<OBJECT ID="MzCOM" CLASSID="clsid:A3B0AF9E-2AB0-11D4-B6D2-0060089002FE">
</OBJECT>

<SCRIPT LANGUAGE="Javascript">
<!--
MzCOM.Eval('(require-library "qq.ss" "quasiquote")')
MzCOM.Eval('...')
//-->
</SCRIPT>
```

Figure 1: Calling MzCOM from Javascript.

tants of some Scheme types, such as fixnums and strings, can be represented as inhabitants of COM types, others cannot. For instance, PLT MzScheme allows users to create structure types with `define-struct`; there are no corresponding types in COM. Two other reasonable design options were considered for `Eval`:

1. return (the COM equivalent of) `void` always — all evaluation is for side-effect only, and
2. return a COM `VARIANT`, a tagged union, such that the union contains a COM value corresponding to the Scheme value returned by the evaluator, when possible, and `void` otherwise

We rejected option (1) because it throws away potentially useful information. We rejected option (2) for the same reason, and also because programming with the tagged union is messy.

To notify its clients of Scheme errors, MzCOM offers an "outbound" interface `_IMzObjEvents`. The `HRESULT` returned by COM methods is only a numeric code, so it is of limited utility. The MzCOM outbound interface has one method, `SchemeError`. A handler for this event is passed a string describing Scheme errors in detail, the same string one would see in MzScheme.

Let us see how to use MzCOM from some different programming environments. For our examples, let's suppose we want to use Shriram Krishnamurthi's `quasiquote` library for PLT Scheme to download stock prices from the World-Wide Web.

MzCOM from Javascript. To use MzCOM in Javascript, we create an instance using `OBJECT` tags, as shown in Figure 2. The `CLASSID` attribute of the `OBJECT` tag specifies the CLSID that identifies the required COM class.

MzCOM from Visual BASIC. Using MzCOM from Visual BASIC is also relatively simple. We use a function to insert double-quotes inside a string:

```
Dim schemeObj As MzObj
Private Function Quote(s)
 Quote = CHR$(34) + s + CHR$(34)
End Sub
Private Sub Form_Load()
 Set schObj = New MzObj
 schemeObj.Eval("(require-library " +
              Quote("qq.ss") + " " +
              Quote("quasiquote") + ")")
 schemeObj.Eval("...")
End Sub
```

Before compiling this program, we need to indicate to Visual BASIC that we are using the "MzCOM Type Library" by choosing it from a list accessed through VB's menus. That choice defines the type `MzObj` for our program.

MzCOM from MysterX. Of course, there is no need to use MzCOM from MysterX, since a MysterX user already has access to PLT Scheme. We show how to do so for the sake of completeness:

```
(require-library "mysterx.ss" "mysterx")
(define mzcom (cci/progid "MzCOM.MzObj"))
(com-invoke mzcom "Eval"
  "(require-library \"qq.ss\" \"quasiquote\")")
(com-invoke mzcom "Eval" "...")
```

The *ProgID* `MzCOM.MzObj` is a string that identifies the MzCOM class via a Registry entry.

Languages that support COM have various ways of setting up COM event handlers. In C++, registering a COM handler for requires a tremendous amount of setup and code, much more than can be shown here. Registering event handlers in Javascript and Visual BASIC is somewhat less involved. In the next section, we shall see that in MysterX, setting up an event-handler for COM classes is quite simple.

3 MysterX

MysterX is supplied as a PLT Scheme *unit*, written in Scheme. The unit loads three dynamic-link libraries (.DLL's), written in C++. Those libraries are linked against the Win32 COM support code. Some of the C++ code also makes use of Microsoft's Active Template Library (ATL).

For the programmer, MysterX has two distinct aspects, a COM aspect and a browser aspect. Some COM classes, such as ActiveX controls, require a *container* for display. To accomodate them, MysterX provides a programmable Web browser that can host COM objects. The browser supports Dynamic HTML (DHTML), allowing the programmer to change the appearance of documents. DHTML elements also generate events, which can handled by Scheme procedures.

3.1 MysterX COM support

Instances of COM classes can be created with `cci/progid`, as we saw in the last section. Each COM class also has a name in the Windows Registry, so MysterX provides an alternate procedure `cci/coclass`. These creation procedures are used when a COM class handles its own visual display, or when its display is unneeded. By default, both of these procedures create instances on the machine where MysterX is running. The optional argument `'remote` creates the instance on a remote machine as specified by an entry in the Registry. The instance can also be run on a particular machine by supplying a machine name. MysterX's remote execution capability relies on Windows' Distributed COM (DCOM) subsystem.

MysterX allows programmers to interactively query COM objects for their methods, properties, and events from Scheme. Unlike other COM implementations for functional programming languages, like those for Haskell [JML98] and O'Caml [Ler99], MysterX does not require a preprocessing step to generate stub code from IDL files containing type information. Instead, type information is obtained dynamically from COM objects, consistent with the spirit of Scheme.

The underlying COM interfaces are hidden. Instead, these attributes are associated with the COM object itself. For example `(com-methods obj)` returns a list of strings that name the methods of the COM object `obj`. Similarly, we have

- `(com-get-properties obj)`
- `(com-set-properties obj)`
- `(com-events obj)`

Each of these procedures returns a list of strings.

Names alone are not enough to use methods, properties, and events. MysterX also allows the *types* of these to be obtained from Scheme:

- `(com-method-type obj "methodName")`
- `(com-get-property-type obj "propertyName")`
- `(com-set-property-type obj "propertyName")`
- `(com-event-type obj "eventName")`

Each of these procedures returns a list of symbols. For instance, a method might have the Scheme type `(string int -> bool)`, which is a transliteration of its COM type. All events have a `void` return type, meaning that the event's handler return value is ignored.

With the information provided by these reflection mechanisms, we can make use of a COM object from Scheme. To call a method:

- `(com-invoke obj "methodName" arg1 ...)`

To read a property:

- `(com-get-property obj "propertyName" arg1 ...)`

The procedure `com-set-property!` sets a new value for a writable property. An event handler `f` for a particular event is associated with an object with

- `(com-register-event-handler obj "eventName" f)`

Returning to our example from the last section, we can easily create an event handler for MzCOM loaded into MysterX with:

```
(com-register-event-handler
  mzcom "SchemeError"
  (lambda (s) (printf "~a~n" s)))
```

3.2 MysterX browser support

If we wish to display COM class instances that require a container, we create instances of the `mx-browser%` class. When created, the browser displays a blank page. We can point the browser to particular Web pages with the methods `navigate`, `go-back`, and `go-forward`.

At any moment, a MysterX browser contains some *document*, which roughly corresponds to the notion of document in the Document Object Model (DOM) standard of the World-Wide Web consortium [Con99]. Each browser navigation results in a new document. Methods in the `mx-document%` class are used to insert HTML into a document.

Individual HTML elements can be retrieved from a document. HTML elements are instances of another class, `mx-element%`. Methods of that class can be called to change elements' style, such as size and color. As specified in the DOM standard, HTML elements generate events, distinct from COM events, such as in response to mouse clicks. With MysterX, Scheme handlers can be associated with the events generated by HTML events.

Underneath the hood, MysterX's browser is Internet Explorer used as a COM component. The MysterX browser can be used either as a browser *per se*, or for displaying formatted text, as in the talk "slides" accompanying this paper.

4 Conclusions

We have presented PLT software that sits at either end of the COM channel: software that makes PLT Scheme available as a COM component, and software that allows PLT to use COM components.

Using Scheme for COM applications has great practical value. In our department at Rice, we have built software using MysterX to transfer accounting data between Excel spreadsheets used for accounting. We intend to use MysterX for validating the contents of local Web pages.

MzCOM and MysterX source code and binaries are available at

`http://www.cs.rice.edu/CS/PLT/packages/`

References

[Box98] Don Box. *Essential COM.* Object Technology Series. Addison-Wesley, 1998.

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

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

[Lei99] Daan Leijen. ScriptServer Web page, November 1999. `http://haskell.cs.yale.edu/haskellscript/scriptserver.html`.

[Ler99] Xavier Leroy. CamlIDL Web page, March 1999. `http://caml.inria.fr/camlidl/`.

[Ste99] Paul A. Steckler. MysterX: A Scheme toolkit for building interactive applications with COM. In *Proc. Technology of Object-Oriented Languages and Systems 1999 (TOOLS 99)*, pages 364–373. IEEE, August 1999.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* ACM Press/Addison-Wesley, 1998.