A Substrate for ActiveX Support in Research Languages

Lars Bergstrom (lars@cs.nwu.edu)

June 13, 2000

Abstract

Most research language implementations are designed to either test out a new implementation technique or language theory. Therefore, they tend to suffer from a lack of libraries for performing common real-world tasks such as database communications, text-to-speech support, and the display of graphical user interfaces. This work attempts to solve this problem by providing a substrate that implements most of the low-level code needed to host ActiveX components, but which is portable between language runtimes with minimal requirements on both the language implementation and implementer.

1 Introduction and Motivation

There are many complaints commonly pointed at common research languages today, the most notable of which is the lack of usefulness in a more production-code-oriented environment. This complaint is one of the major barriers to entry for most languages, effectively crippling them against languages such as C, Visual Basic, and Java, which have very powerful, mature libraries and tools available. The substrate presented here allows any research language that has a rudimentary out-call C foreign-function interface to easily leverage all of the power of ActiveX controls.

ActiveX controls are binary modules that follow a component standard put forth by Microsoft in order to provide a consistent interface to functionality. ActiveX controls are the core foundation of functionality in languages such as Visual Basic, the world's most widely used language [Chappell]. Because the majority of the ActiveX controls used in Visual Basic applications are free, leveraging this functionality gives any research language that runs on the desktop Windows platforms (Windows 9x, NT, or 2000) the ability to provide applications just as rich as those produced by standard 'commercial language implementations'. This substrate allows language implementations to use these components and to take advantage of changes in the operating system without either polluting the core language with an unnecessary number of new primitives or requiring extensive modification of the underlying language runtime. In an extreme example, this substrate could be integrated in such a way that it would expose only two new primitives into the target language, while still capturing most of the power of ActiveX.

We first give a general overview of ActiveX and COM that covers all of the major topics needed to understand the functionality implemented and exposed by this substrate. Then, we discuss the architecture of this substrate, in order to provide interested parties with an overview of its exposed functionality. The focus then shifts to explain the tasks ahead of any language implementer considering embedding this substrate within their current language infrastructure. Throughout, WinScheme 48[1] is used as an example language in which we have embedded this substrate. Finally, other ActiveX and COM-supporting dynamic languages are presented and remaining issues and possible room for future improvements in this substrate are discussed.

2 Overview of ActiveX and COM

This introduction to COM and ActiveX is sufficient only in that it explains the bare minimum necessary to use this substrate effictively. If you actually wish to become proficient in COM, you are highly encouraged to obtain the resources listed at the end of this paper. If you are already proficient in COM, feel free to simply skim the following text to make sure that you are completely familiar with the concepts of most importance to this implementation.

2.1 General COM terminology

Before you can answer the question of what ActiveX is and how it works, it is essential to know what COM is and how it works – ActiveX is a subset of COM. COM stands for Component Object Model and was designed by Microsoft as a solution to the problem of binary-level component reuse. It is a technology enabling developers to program against and implement interfaces, which are basically abstract definitions of the ways in which you can interact with an object. For simplification, you can think of them as either C++ pure virtual class declarations (without any implementation!) or Java interfaces.

The nice thing about COM interfaces is that they are immutable once published. You can provide different implementations of an interface, but if you want to

[1] WinScheme 48 is a Windows port of the Scheme 48 system developed by Richard Kelsey and Jonathan Rees.

add a method or property to an interface, that requires the definition of a new interface with a new name. You may use inheritance to carry functionality between versions, but once an interface has been created and published, it shall never change.

Client programs use components by requesting the loading of an implementation object and then using the Microsoft-defined interface IUnknown to determine supported interfaces. The general idea is that your programs will understand a set of well-known interfaces and that you will let the user change the class object being used. To give a trivial example, one might define a spell-checking interface that your application would use. This interface would be documented and included with the development kit for your program, and your program would likely come with a default implementation. With the development kit, other developers would then be able to precisely follow your interface but perhaps implement a spell-checker with different runtime characteristics or language support. They could even implement one which simply wrapped yours, adding support for technical words. Once this component is written, the user simply installs the third-party component and switches the current object to use. There is no need for a recompile, re-release of your application, or any other intervention on your part.

2.2 COM Naming

Collision of names of variables, functions, and other language constructs is a common problem. Various solutions have been proposed, varying from the namespaces of C++ to the module system of scheme. COM, however, takes a different approch. Using Universally Unique Identifiers (UUIDs or GUIDs), all names exposed from a COM component are instead translated into 128-bit numbers. Mathematical guarantees are made at the time of the generation of these numbers that basically ensure that they will never collide. Since these are so useful – avoiding any need for additional language constructs while still solving the naming problem – they are used in many situations within COM. In each case, they are referred to by a different name:

- Interface identifiers (IIDs). Interfaces are typically defined in text by concatenating a capital "I" with the word or words denoting the interface. For instance, a spell checking interface might be named ICheckSpelling. Programmers may use the friendly name of ICheckSpelling, but behind the scenes, it will always be treated as an IID.

- Class identifiers (CLSIDs). Classes are the name of the objects which actually implement a set of COM interfaces. They are typically named by a concatenation of words. For instance, the object implementing a spell checking interface that uses a certain well-known and extremely large dictionary might be named OEDChecker.

- Library identifiers (LIBIDs). Type libraries are generally named as class identifiers are; simply concatenate the words. Continuing the spell checker example, the type library's programmer-friendly name might be SpellOMatic.

This solves the naming problem; however, it failes to address the problem of simply wanting to instantiate the latest and greatest version of a component installed on a machine. Thankfully, there are programmatic identifiers (ProgIDs). These are simply friendly strings which are associated in the Windows registry with the CLSID of an object to load. For instance, the ProgID for Internet Explorer is "InternetExplorer.Application", and has been for many versions of the browser, despite the fact that the CLSID has changed with each release.

2.3 General Automation and ActiveX terminology

Automation is an umbrella term, encompassing all applications that produce and/or consume ActiveX objects. An ActiveX client is simply a program that uses COM to instantiate (host) ActiveX objects. An ActiveX object is the thing provided by an application that implements and exposes functionality. Note that an application can and very often does consist of both. In that case, the application is referred to as either an ActiveX object or an ActiveX client, depending on the role in which it is currently participating [Automation].

While the absolute-minimum, most-trivial definition of an ActiveX object (sometimes called also called component or control) is simply any COM object which supports IUnknown and can register itself on the user's system, they usually also:

- expose the IDispatch interface
- have type library information
- use only OLEAUTOMATION-compatible types
- notify their owner of events and state changes

Some ActiveX controls also provide a graphical interface, though we will describe this feature only in passing.

2.4 The IDispatch interface

The IDispatch interface is another Microsoft-defined interface (derived from IUnknown, as all interfaces must be). It supports basic functionality for abstractly calling functions and getting or setting properties through use of an integer-valued dispatch identifier (DISPID). These DISPIDs can be determined by the client through either a function on the IDispatch interface or by reading the type library information.

The primary method defined by the IDispatch interface is called Invoke. Among its arguments are the DISPID of the method to call (or property to get / set), an array of arguments to pass in, and a place to store the return value. It is easy to see how, given a full description of the methods, types of arguments, and types of return values of the object being implemented, a runtime could easily access all of its functionality.

Two additional methods allow the client to get the DISPID for a method or property based upon the string form of its name and to get its associated type information.

2.4.1 Properties and methods

Properties and methods are the two classes of functionality that are exposed from IDispatch-derived interfaces. Properties are conceptually data associated with an interface. They may be readable, writeable, or a mix of the two. In general, it is considered bad form for retrieval of a property to side-effect the state of an object. For example, an ActiveX object implementing a readable NextElement property should not remove the retrieved element from an internal list of elements or update an index being kept into the list. If retrieved twice in a row without any intervening property-writes or method-invocations, property reads should be identical.

Methods are normal functions which take arguments and return results. ActiveX allows the return of multiple values, but only one will be marked the primary return value. It is the one which is usually returned as the result of a method invocation in higher-level languages – all other return values are usually treated as by-reference in paramaters which are side-effected as a result of the computation.

Both methods and properties are implemented through calls to the Invoke method of the IDispatch interface. Because of this, it is actually the choice of the high-level language implementor whether or not to differentiate between them; however, no common higher-level language that abstracts ActiveX fails to do so.

2.5 Type information

The need to distribute ActiveX controls to others and allow them to easily use the control from any programming environment resulted in the creation of a binary type information format. Type libraries are stored in an undocumented format, but Microsoft provides a set of interfaces and functions for both reading and writing type libraries. These libraries contain all of the information about the objects, type aliases, and interfaces (and their associated methods & properties) implemented by an application.

ActiveX clients which load this information and then directly call methods and properites on ActiveX objects without querying for the DISPIDs of the names of the properties and methods at each invocation are called early-bound clients. Late-bound clients query at every use of a property or method for its proper DISPID. Some clients use the type library information to make raw calls against the C vtable of non-IDispatch-derived interfaces, given sufficient information. These clients are called either vtable-bound clients or simply C-bound clients, though there is much incorrect documentation in existance that also refers to them as early-bound.

Tools such as OLEView (provided by the free Microsoft Platform SDK) and the Object Browser (included with Visual Basic) allow users to browse the information embedded within type libraries. Type libraries are usually either built directly into the redistributable control (usually either a .dll or .exe file) or are packaged separately (as a .tlb file).

2.6 OLEAUTOMATION types

OLEAUTOMATION types are simply a restricted set of variable types (i.e. long, short, currency). Most ActiveX controls designers restrict themselves to this set of types when building controls. The type library information also can immediately tell a runtime whether or not a particular control is OLEAUTOMATION-compliant. These types are often referred to as VARIANTs, as that is the name of the C data type which is a structure combining a tag denoting the OLEAUTOMATION type and a union of all possible OLEAUTOMATION types (into which the value is actually placed). VARIANTs are the actual types which are used when calling the Invoke method of IDispatch.

Restriction of ActiveX objects to these types means that an implementation with the ability to utilize those types will have full access to all of the functionality of the object. While, naturally, you lose C's ability to pass around void-pointers and, of course, you lose scheme's ability to pass around continuations, you still have not only items such as numerical and string values, but also user-defined record types, enumerations, pointers to ActiveX objects, and dates, to name a few. An up-to-date table of all of the types and their exact sizes and formats can be found in the Microsoft Developers' Network Library [VARIANT].

2.7 Events

Notification of event and state changes is handled through two separate mechanisms in ActiveX. If a control wants to allow its host to be notified of changes of internal state (properties), it will allow the host to implement the Microsoft-defined IPropertyNotifySink interface and register it with the control's connection point container. Connection points are beyond the context of this paper, but suffice it to say that they allow hosts of objects to provide the controls being hosted with a way of calling back into the host. They are theoretically not limited in the number of registered interfaces (commonly called sinks, whereas the control is sometimes refered to as the source, in this context).

ActiveX also allows controls to define rich event interfaces for their hosts to provide to them. For instance, one version of the Microsoft XML control's XMLDocument object allows the host to define an interface with two methods: one signaling that data is available and the other that the parser has moved on to a different stage of processing the XML document being loaded. These event interfaces are stored in the type library information and any suitably-motivated runtime can traverse the information and discover which interfaces to implement.

At this point, the following question is begged: how can a runtime, which has only just learned about an interface, provide an implementation of it for registration with the ActiveX control? Luckily, no non-trivial dynamic creation of interfaces is truly neccessary – all event interfaces are required to be derived from the IDispatch interface. Therefore, all that a host implementation needs to do is to define an implementation of the IDispatch interface which can masquerade as an interface that it learns about at runtime (through the type library information) and pass an instantiation of that IDispatch-derived interface to the control.

2.8 Objects

Since interfaces are merely contracts of method definitions, an additional construct is needed to denote implementation. A COM class (also called coclass for short) is an object which implements an interface or set of interfaces. A client which wants to use an interface actually goes through the following steps:

- Create an instace of the factory object which implents the desired COM class

- Request creation of an instance of the object (initially in the form of an IUnknown pointer)

- Request the interface upon which to invoke methods (through the use of the QueryInterface function on IUnknown)

The CoCreateInstance method does all of these steps as one, but it is helpful to know the internals, as it makes both debugging and more advanced concepts much clearer.

A coclass can also have an associated set of event interfaces. These are not actually interfaces which it exposes, but rather are the custom interfaces metioned in 2.7. There is an idea of a default event interface, which is the primary interface ActiveX clients are expected to implement to receive events. However, coclasses can have an unlimited number of event interfaces, whether for backwards compatibility with old versions of the object or for proper factoring of interface functionality.

2.9 Hosting visual controls

ActiveX controls are the current incarnation of a long line of specifications for object implementations. Originally, it all started out with Visual Basic Extensions (VBXs), which were primarily graphical controls that a user could plug into a Visual Basic form and use. Examples of these include the standard windows pushbutton, drop-down listboxes, and tree-view controls.

As such, the ability to provide graphical interfaces has a long and complicated history. That history is outside the context of this paper. All that needs to be known here is that the substrate which has been implemented here does not support hosting controls which provide graphical user interfaces (or, at least, does not allow them to display themselves). It does, however, allow the user to load the controls into the Microsoft Internet Explorer 5.0 or greater web browser and still allows the same level of functionality (calling methods, getting and setting properties, and handling events) that would be expected of a standard, non-graphical control.

For more information about the history of graphical interfaces and a general overview of the internals thereof, see [Chappell] and [Brockschmidt].

3 General Architecture

At this point, it is helpful to see the desired result from the integration. While, of course, database code and graphical user interfaces are motivating reasons to integrate ActiveX support, no demo seems to motivate people more than the fact that the Microsoft Agent can be forced to do our bidding. The following sample code loads the Clip-It agent and causes it to say something witty and appropriate:

```
;; Load the Microsoft Agent infrastructure
(define agent-control
  (create-object "Agent.Control.1"))
(put agent-control "Connected" #t)
(define agent-characters
  (get agent-control "Characters"))
;; Load and select the Clip-It Agent
(invoke agent-characters "Load" "clippit"
```

```
  "C:/program files/microsoft office/office/clippit.acs")
(define clip-it
  (invoke agent-characters "Character" "clippit"))
;; Mua-ha-ha! Now it must do our bidding!
(invoke clip-it "Show")
(invoke clip-it "Speak"
  "I may be a twisted, evil piece of metal... but at \
   least I don't have to write research papers!")
```

Create-object loads an ActiveX component into scheme. Get and put are two methods for reading and writing component properties, respectively. Note that, as shown in the definition of the agent-characters variable, in addition to standard data types, other ActiveX objects can be returned as the value of properties or from method calls. Finally, the invoke scheme method calls an ActiveX method exposed from the object. In this case, the Agent is made visible and then commanded to speak.

3.1 Overview

There were two primary goals in this substrate's design. First, it had to be easy to understand and use for researchers exposing an out-call only C language foreign function interfaces from their underlying language implementations. This substrate should be possible to add in a day or two by an experienced C coder familiar with the inner workings of the research language's implementation, but whom might have a very limited amount of experience with COM.

Second, this could not require that people implementing dynamic languages force their users to create 'stubs' or perform other akward processes. Many people use a dynamic language because they allow extremely rapid prototyping and debugging. Forcing a user to create files that then are linked back into the running interpreter – even just once per object – goes completely against this entire culture. While in many cases generating stub files would make things somewhat easier for the implementer of a substrate or could even stand to increase performance, it introduces inexcusable inconsistencies in development methodologies for the user of dynamic languages.

3.2 Detailed Substrate Description

3.2.1 Object loader & unloader

The object loader uses the ProgID, to identify the binary ActiveX object to load into the language runtime. If it failes to load the object, this is appropriately signaled to the language; otherwise, a value representing the object is returned.

The substrate also provides hooks for the destruction of objects and reclamation of their resources when the language implementation determines they are no longer needed. These hooks may either be called directly by end users or may be installed into any language's garbage collection system.

3.2.2 Methods & properties

The substrate handles taking an object returned by the object loader, the ID of the method to invoke, and a series of variables defined in the language implementation and does:

- checking of parameter count
- automatic fill-in of 'default values', if requested by the language
- automatic parameter type conversion (i.e. the string "123" to the integer '123')
- provision of verbose error information through the COM exception mechanism, when user-level errors occur
- execution of the action
- proper return of the result, if the execution completed successfully

3.2.3 Events

The event support can be broken into two main parts: property change notifications and control-defined events.

Property change notifications are supported by the substrate through the implementation of the IPropertyNotifySink interface, a queue for the notifications as they come in, and a rich way for languages to retrieve notifications.

Control-specific event interfaces are supported by the substrate. The language implementation simply calls down into the substrate, requesting an object to handle events on and the name of an event interface, and the substrate handles all queuing. The names of all possible event interfaces and their associated event methods is returned during the object creation stage through the type library introspection.

Neither of these relies on any support in the language for native OS-level threads; indeed, they could be used in a language which had no threading support at all. However, if you do have threads (either native or language-level), there is additional functionality that may be used to implement non-blocking, highly-efficient handling of the asynchronous events and notifications.

The event substrate automatically handles all pathological cases such as object removal without event interface removal, associated resource handling, and overflow of event messages due to neglect to service.

3.3 A more involved example

The following WinScheme 48 code loads Internet Explorer, browses to the author's homepage, and then sets up an event handler to notify WinScheme asynchronously if the user leaves that page and to take appropriate action to deter such behavior in the future:

```
;; Load Microsoft Internet Explorer
(define ie
  (create-object "InternetExplorer.Application"))
(define ie-source-iid
  "{EAB22AC2-30C1-11CF-A7EB-0000C05BAE0B}")
;; Browse the The Wonderful World of Lars
(invoke ie "Navigate" "http://www.lars.com/")
;; When the user goes to another page, react!
(register-object-event-handler!
  ie ie-source-iid
  "BeforeNavigate"
  (lambda (a b c d e f)
    (format #t "You left my home page! Reformatting \
                hard disk...~%")))
;; Load IE on-screen
(put ie "Visible" #t)
```

This sample uses many of the same primitives as seen in the earlier sample; however, it also uses the register-object-event-handler! method. This method allows the user to specify a user-defined method to call whenever a specific event on an object is fired. Note that the WinScheme 48 implementation of the substrate requires the user to pass in the IID of the event sink to implement – the substrate also allows that to be unspecified, which instead means that events would automatically be handed on the default event interface. While this is often sufficient behavior, as many object have but one event interface, there are some objects which support several event interfaces whose functionality does not overlap the default's.

4 Creation of a language-level interface to the substrate

This section of the paper describes the code that must be written by a language implementer in order to use the substrate. There are, of course, many ways of wrapping this substrate and exposing ActiveX functionality to end-language-users – the approches used in WinScheme 48 should be used only as one possible example.

4.1 Type library information

Type libraries provide all of the information necessary to perform method invocations and property manipulation. This information must be loaded into the runtime, as it provides the mappings from user-friendly method and property names into the integer values that the substrate requires.

Two functions are provided to retrieve this information. One takes the name of a file containing type library information and is useful for large libraries which could potentially take a long time to load (such as the one describing Internet Explorer, which is over 8 MB). The other takes a library ID and version number, which are easily retrieved through a utility function, executable on a running instance of a COM object. This allows the language implementer to load objects and then only load type information when it is actually required.

To make it easier for C language programmers, bindings for all of the type library information are provided in the form of simply C structures. There is also a deletion function which should be called when you no longer need the structure.

In WinScheme 48, we provide a wrappers for loading type library information by path for people who wish to load a large amount library into memory and then save it off to disk. When the type library structures are returned from the C layer, we use WinScheme 48's VM allocation routines to turn the structures into scheme lists containing all of the information and then allow the substrate to reclaim the memory.

4.2 Wrapper for object loading and unloading

The substrate provides all object loading and unloading functionality. Simply call the create function with a string denoting the COM ProgID of the object to load and it will return to you a pointer to an IDispatch interface. This will be a 32-bit value (though, in truth, an implementation need only keep the first 30, since Intel machines are word-aligned). It is the responsibility of the language implementation to keep track of this value, as it will be passed in to every other function in the substrate.

In WinScheme 48, we allocate a language-level structure to hold it. The virtual machine then allows us to connect a finalizer to the object. A finalizer is a procedure to call when the garbage collector determines that there are no more extant references to the object; this finalizer ends up being basically just a call to the substrate-provided object deletion routine.

4.3 Utility functions

There are three important object introspection utility functions implemented by the substrate. One gets the LIBID and version of the COM type library that provides all of the information about the object. Another retrieves the IID of the interface through which you are communicating with the object. Use this IID to traverse the type library information and determine the methods and properties exposed by the object.

The final utility function retrieves the IIDs of the event interfaces. Event interfaces are generally object-specific, though some objects do share a common event interface. These interfaces' methods are described in the same type library as the object's actual class and interface information. Since objects can support multiple event interfaces, this utility function can return any number of event IIDs (including zero).

WinScheme 48 simply wraps all of these functions, though they are not user-exposed. They are used internally in things such as the method/property invoker, event notification registration functions, and object display mechanisms.

4.4 Methods and properties

As alluded to before, methods and properties are actually implemented by the same code base in the substrate. A good rule of thumb is that methods take and return values, readable properties return values, and writeable properties store values. There are a couple of violations of this, such as methods with 'void' return types and properties representing arrays (in which case the read-property takes an additional parameter – the index or indices into the array!), but it is true in the general case.

The responsibility of the language implementer is to provide a mechanism for invocation. If the user is allowed to use friendly names for methods and properties, the language bindings will have to look up the object's type information to determine the ID corresponding to the friendly name of the method or property. The language bindings must also inform the substrate of whether the operation is intended to be a method invocation, property-read, or property-set. After that, though, the arguments may simply be passed into this substrate, which will handle all necessary type coercions and error handling.

There aren't too many options for the implementation of this. WinScheme 48 exposes get, put, and invoke functions at the language level. The first parameter to each is the object on which to operate. The second is the name of the property or method; the additional parameters are all optional and are simply passed straight down into the language substrate. The language substrate is then used to inform the user of all type errors, though in truth the loaded type library information could likely be consulted before invocation to provide slightly richer error messages, as those provided by the substrate are merely positional, providing no information about the expected or provided type.

4.5 Registration of event wrappers

The two classes of events are notifications of property changes and custom events. Notification of property changes is supported by simply calling substrate function on the object with an integer identifying the object (more on this later). This function returns an id (or 'cookie') for the registration, which is then passed to the destruction function when the connection no longer needs to be maintained. Custom events are analogous, with the addition of a single parameter – the ID of the custom event interface on which to handle events.

Event notifications are handled asynchronously by the substrate without any interaction on the part of the VM. All COM events are queued and indexed by the id of the object. The id of the object itself should be generated by the language runtime and is simply constrained by being a positive integer value.

Processing of events can be handled in two ways. The simplest is a set of thread-safe functions for checking the state of the queues (both notification and custom event) and for retrieving C structures representing the pending events. In theory, a language implementation without support for threads or any other form of parallelization could simply write a process-pending-events function that processed any pending events and then returned.

The second possible method involves the use of a named Win32 event handle. Win32 event handles are operating system objects that work somewhat like old mailboxes – they are either flag-up (signaled) or flag-down (unsignaled). A language runtime can spawn off threads (language-level or OS-level) to use the Win32 wait-functions on these objects, which will be set to the signaled state every time that events are placed on the notification and custom event queues by this substrate. Once those go off, it is safe to attempt to retrieve the raw C structure serialization of the ActiveX event notification without fear of blocking.

The WinScheme 48 version creates an object id at object-creation time. This id is then used for both notification and custom event handlers. The WinScheme 48 virtual machine was previously modified to natively use Win32 object handles as its primary means of non-blocking I/O, much akin to how many Unix-native language runtimes use file descriptors. Therefore, since events are a class of object handle, it was trivial to hook in event support to the virtual machine, spawn off a language-level thread (which is not a native OS thread), and use that to implement all of the event handling.

Users are allowed to register handlers for individual methods, instead of for the interface on the whole. The backend language runtime naturally hooks all of the methods (COM would not allow otherwise!) and keeps an in-memory table of the functions which the user has set to handle the incoming events, if any. Non-handled events are simply discarded.

One final note about the event mechanisms is that they can be completely ignored. If a language implementation is only ever going to use ActiveX to access controls which can be set into synchronous execution mode (both the database and XML controls fall into this category), then the event infrastructure can be left unhooked for initial prototypes. It could even be left out entirely, though it would severely restrict the ability to build GUIs.

4.6 Conversion of native types

COM achieves a large portion of its language interoperability through its type system. There is a set of types, known as OLEAUTOMATION types, which are the set of valid variable types in ActiveX. The major task for a language implementer is to write a pair of functions that map to and from the language's native types and the COM OLEAUTOMATION types. It can also be helpful to write a function that takes the integer values denoting the different COM types and returns the analogous type in the language. This is only required if the language runtime is going to display type information to the user and has the benefit of being easily implemented within the target language (rather than in the C runtime).

WinScheme 48 cheats a bit in this aspect. First, there are types for denoting both dates and currencies within COM. Since there is no scheme language analogy for those types (ignoring the ongoing SRFI process for standardizing them), we simply do not support them, though we do allow them to be 'boxed' and passed through. Basically, if one is the return value from a function or a readable property, it can be passed in as an argument to another function call or as the value for a writeable property – but it cannot be natively read or modified within WinScheme 48.

Coercion of types, too, has a loophole that makes it easier than one might initially think. Since the scheme language is basically untyped, we simply coerce the value passed into the function into any convenient type based upon its runtime value. Then, the substrate will automatically try to coerce it. If the user passes in the string "123" and we coerce it to an OLEAUTOMATION string and pass it in to a function expecting an OLEAUTOMATION integer, the substrate itself will handle that coercion. Therefore, the type coercion macros (in both direction) have no knowledge at all of the target type of the operation.

4.7 Final bits of glue

There are two final tasks that should be handled by the language runtime. Two functions are exposed which initialize and reclaim resources associated with the handling of events (allocation of the queues, creation of event handles, etc.). These should be called at the start and end of execution sessions.

WinScheme 48 simply calls the initialize function the first time that the user requests to add either a notification or custom event handler. The reclamation then happens when the environment is closed down.

5 A look at prior art

Here, we look at several other approaches to the addition of ActiveX support within dynamic languages. While all of the source available was too tightly tied to the language implementations to be of any use for a common language substrate, it is still useful to see the level of ActiveX / COM support available in other languages. We give overviews of two research language implementations, two open source language implementations, and a commercial language implementation, respectively.

5.1 HaskellScript

HaskellScript is an implementation of ActiveX support within the Haskell language [Finne]. This implementation consists of two main components: the foreign-language hooks within the Glasgo Haskell Compiler (GHC) runtime and H/Direct. The foreign language support in GHC allows use of enough C code to wrap COM natively from Haskel code. However, writing code at this layer would be inconvenient for programmers, so they also developed H/Direct, a tool for transforming MS IDL into Haskel programs. After using the tool, a user may simply program against the stub functions to create and use COM objects.

Since their native language supports in-calls (calls from other languages into the target language), they have also address a problem that this substrate does not – that of exposing COM objects in the native language for use by the outside world.

There is also an ActiveX Scripting Engine available for Haskel. This allows Haskel code to be used directly within web pages and Windows Scripting Host Components, just as VBScript and JScript are.

5.2 MysterX

The Rice PLT group has also done an implementation for their scheme interpreter, mzscheme. This library is known as MysterX [Steckler]. Their implementation offers many of the same features of this substrate; however, it works by hosting a web browser control within a window associated with the application. Once one of these windows is created, the user then loads ActiveX controls into it through the use of the DHTML object model. This is true for both visual and non-visual ActiveX controls.

5.3 ActiveState Perl

The Win32::OLE package within ActiveState's Perl implementation provides access to much of the same functionality which this substrate does [ActivePerl].

Their support for asyncronous events was not yet finalized at the time of writing, but was due to be released shortly.

5.4 Python for Windows

Python allows users to both use COM objects as a client and expose COM objects from a Python-based server [Hammond]. Their client package provides two mechanisms through which to access COM objects. Dynamic invocation allows stubless use of ActiveX objects from the language, provided that all method names and types are known. There is also a tool packaged with the distribution which creates stub files to ease programming and generate a form of documentation for the interfaces and objects being used.

At the time of the writing, the author of PythonWin had just moved to ActiveState and was about to release a new version, named ActivePython. It was supposed to contain many additions to the COM support.

5.5 Visual Basic

Microsoft's flagship development product, Visual Basic is built for COM support from the ground up [VB]. The entire language uses COM and ActiveX when building forms (graphical user interfaces) and for all of the controls and routing mechanisms. Users can both create and consume ActiveX controls directly from the development environment without the use of external tools.

Using ActiveX objects can either be done dynamically by programmers with foreknowledge of the object in question or through the use of the type library. Once Visual Basic loads a type library, it can, in some cases, actually forego the slower IDispatch-based method invocation and use raw C virtual function tables to invoke methods which it has only just learned about, through the judicious building of stack frames by hand in assembly.

6 Future plans

As art, no piece of code is ever truly finished – only abandoned. While completely functional in its current state, there are naturally a few additions that could stand to make the implementation more attractive to language implementers.

The substrate itself could use:

- More sample language runtimes, for other languages
- Heavy marketing to other research groups writing language implementations

- A 'porting kit' containing documentation, pre-compiled binaries, and source for use by other language implementers

The WinScheme 48 implementation could use:

- Native support for array types
- Removal of the requirement of loading the type library information (8 MB of Internet Explorer type information is the motivator for this)

7 Conclusion and observations

This substrate is functionally complete. At this point, there is no reason for implementers of other languages to write their own ActiveX language hooks, as any language implementation with even modest C language hooks within its core can take advantage of not only the volume but also the vast testing that has been performed on this code base. All that remains now is to make it more accessible to others in the research community.

References

[ActivePerl] ActiveState Perl. Win32::OLE - OLE Automation Extensions. http://www.activestate.com/Products/ActivePerl/docs.index.html.

[Automation] Microsoft. Automation Programmer's Reference. Microsoft Press, 1997.

[Brockschmidt] Kraig Brockschmidt, Inside OLE, Second Edition. Microsoft Press, 1995.

[Chappell] David Chappell. Understanding ActiveX and OLE. Microsoft Press, 1996.

[Finne] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. Calling Hell From Heaven and Heaven From Hell, Presented at ICFP '99.

[Hammond] Mark Hammond and Greg Stein. Python and COM. http://www.python.org/windows/win32com/COMTutorial/index.htm, Presented at the 6th International Python Conference '97.

[Steckler] Paul Steckler. MysterX: A Scheme Toolkit for Building Interactive Applications, Presented at TOOLS '99.

[VARIANT] Microsoft. VARIANT and VARIANTARG. http://msdn.microsoft.com/library/default.asp?URL=/library/psdk/automat/chap6_7zdz.ht

[VB] Microsoft. Visual Basic. http://msdn.microsoft.com/vbasic/.