Connecting Lisp-Stat to COM

Luke Tierney *

January 10, 2000

1 Introduction

Statistical software systems are just one, albeit a very important one, of several software systems used by data analysts. Data to be analyzed often resides in data bases or spread sheets; results of analyses are often presented in reports written in word processors or report generators. In recent years several mechanisms have been developed for connecting these and other applications, and more generally for allowing software entities to be developed as separate components. The two most important systems are the Common Object Request Broker Architecture (CORBA) [8, 9] and Microsoft's Component Object Model (COM) [2, 7]. While arguably inferior in a number of respects, COM is, in terms of installed base, far more widely used. In particular, most major applications available on the Win32 platform use COM to provide a programatic interface by which they can be controlled by external programs.

This paper describes an interface between one statistical system, Lisp-Stat [12], and COM. The emphasis is on the dynamic subset of COM known at the time of writing as *Automation*. After providing some brief background on Lisp-Stat and COM in Section 2, Section 3 uses a series of examples to show how the interface is used. Section 4 discusses issues in implementation and integration relevant to other statistical systems and other component systems. The final section offers some conclusions and presents some open issues.

Many high level and interactive languages have been interfaced to COM. This work draws mainly on the Perl interface [10, 11] and the Haskell interface, described in a paper with a wonderfully descriptive title [4]. Other languages with COM interfaces include Python [13] and Tcl [14].

2 Background

This section provides very brief outlines of Lisp-Stat and COM.

2.1 A Brief Outline of Lisp-Stat

Lisp-Stat [12] is a statistical computing environment based on the Lisp language. It includes standard numerical statistical operations as well as a customizable dynamic graphics system. The expressions

```
(setf  precipitation
       (vector .77 1.74 .81 1.20 1.95 1.20 .47 1.43 3.37 2.20
               3.00 3.09 1.51 2.10 .52 1.62 1.31 .32 .59 .81
               2.81 1.87 1.18 1.35 4.75 2.48 .96 1.89 .90 2.05))
(mean precipitation)
```

*Research supported in part by grant DMS-9971814 from the National Science Foundation.

for example create a vector containing some data, assign the vector to a variable `precipitation`, and then compute the mean of the vector. As another example, the expression

```
(spin-plot (list hardness tensile-strength abrasion-loss))
```

creates an interactive rotatable three-dimensional plot from a set of three variables.

Lisp-Stat includes an object-oriented programming system that is used in representing many complex entities, including graphs. Messages are sent to objects using the `send` function. The expression

```
(send spin-plot :num-points)
```

for example sends a message to the `spin-plot` object that returns the number of data points in the plot.

2.2 A Brief Outline of COM

In the Component Object Model, objects are servers that expose properties and methods to clients. The servers can run in the same process as the clients or in separate processes, possibly on different machines. COM specifies a language-independent binary interface that allows clients to call objects by a standard mechanism regardless of whether the objects are implemented in process, or in another local or remote process [2]. The dynamic subset of COM called Automation is particularly designed for use from scripting language clients.

Most major Win32 applications expose their functionality as COM Automation objects. This allows them to act as servers to other applications that support a scripting language such as Visual Basic for Applications (VBA). As an example, the VBA expressions

```
Dim wb As Object
Set wb = GetObject("cars.xls")
```

start a new Excel session by opening the file `cars.xls` and store a reference to its workbook object in the variable `wb`. The VBA client can then query or set properties and call methods on this object. For example, the expression

```
wb.Worksheets(1).Cells(1, 1).Value
```

returns the value of the first cell in the first worksheet of the workbook `wb`.

The distinction between clients and servers may be blurred. If a client passes a reference to one of its own objects to a server and the server then calls methods on that object, then the roles of client and server are reversed. One particular area where this is important is in handling Automation events, a mechanism to allow clients to register themselves to be notified when interesting events occur in the server.

The properties and methods of the particular objects exposed by different applications are documented with the applications or in the literature available for these applications. The object interface to major Microsoft applications is described, for example, in [6]. For many applications it is also possible to obtain some information about available objects, properties and methods using an object browser such as the publically available `OleView` [7] utility.

3 Examples

This section presents the Lisp-Stat COM interface through a series of examples. In the first example Lisp-Stat is a pure COM client and in the second it is a pure server. The third example combines client and server features by registering a handler for events. The final example illustrates the use of COM in a distributed environment.

3.1 Lisp-Stat as a COM Client

The primary Lisp-Stat functions used for interacting with COM objects are the function `property` used to query properties and `invoke` for invoking methods. Expressions of the form

(setf (property *object name*) *value*)

can be used to set the property *name* of *object* to *value*.

Several functions are available for obtaining initial object references. The function `get-object` creates an object reference for an object stored in a file. For example, suppose an Excel workbook with several columns of data on cars is stored in a file `cars.xls`. The Lisp-Stat expression

```
(setf wb (get-object "cars.xls"))
```

is analogous to the VBA expression given above. It creates a new Excel process with the specified file and returns a reference to the workbook object.

At times it is useful to separate the steps of starting a server application and loading a data file. The expressions

```
(setf xl (create-object "Excel.application"))
(setf wb (invoke (property xl :workbooks) :open "cars.xls"))
```

start a new Excel session, storing a reference to the application object in `xl`, and then use the `Open` method of the application's `Workbooks` property to load the file. Property and method names are not case sensitive; in Lisp-Stat they can be specified as strings or as keyword symbols, symbols starting with a colon.

The first worksheet in the workbook is obtained by

```
(setf sheet (property wb :worksheets 1))
```

Excel allows the rectangular region containing a specified cell and bordered by empty cells to be obtained by the `CurrentRegion` property of the cell. Thus assuming that all the data are in a contiguous rectangular region, the data can be read from Excel into Lisp-Stat with the expression

```
(let ((range (property (property sheet :cells 1 1) :CurrentRegion)))
  (setf data  (property range :value)))
```

The result contained in the `data` variable is a Lisp-Stat matrix:

```
> data
#2A(("Mazda RX4"     21.0 6.0 160.0 110.0 ...)
    ("Mazda RX4 Wag" 21.0 6.0 160.0 110.0 ...)
    ...)
```

This matrix can then be used with any Lisp-Stat commands; for example a selection of three columns of the data can be placed in a rotatable plot with

```
(let ((cols (column-list data)))
  (setf car-spin (spin-plot (select cols '(1 3 4)))))
```

```
1  Sub SpinPlot(x, y, z)
2      Dim xls As Object, sheet As Worksheet, data As Range
3      Set xls = CreateObject("XlispStat.application")
4      Set sheet = Application.Workbooks(1).Worksheets(1)
5      Set data = sheet.Cells(1, 1).CurrentRegion
6      xls.Visible = True
7      xls.EvalNoValue "(defun sp (x y z)" & _
8                      "  (flet ((as-seq (x)" & _
9                      "           (compound-data-seq x)))" & _
10                     "    (spin-plot (list (as-seq x)" & _
11                     "                     (as-seq y)" & _
12                     "                     (as-seq z)))))"
13     xls.CallNoValue "sp", data.Columns(x).Value, _
14                           data.Columns(y).Value, _
15                           data.Columns(z).Value
16 End Sub
```

Table 1: VBA subroutine to display specified columns of an Excel spread sheet in a Lisp-Stat spin plot.

3.2 A Lisp-Stat COM Server

The roles of Excel and Lisp-Stat in the previous example can be reversed by using the Lisp-Stat application server object from an Excel client. Table 1 shows a VBA subroutine for displaying a specified set of three columns from an Excel spread sheet in a Lisp-Stat spin plot. Line 3 creates a new Lisp-Stat application object, saving its reference in the variable `xls`. Lines 4 and 5 select the data region to be used.

By convention, COM servers typically are started with no visible user interface. To make the server application visible, its `Visible` property must be set to `True`; this is done in Line 6.

The basic objective is to call the Lisp-Stat `spin-plot` function with the data from the specified columns. This function expects a list of three lists or vectors, but data column values produced by Excel are single column matrices. Lines 7–12 use the Lisp-Stat application object's `EvalNoValue` method to define a function that receives the data in a form natural to Excel, converts the data to the form needed by `spin-plot`, and makes the call. The `EvalNoValue` call passes a string argument to Lisp-Stat where the contents of the string are read and evaluated. The value is ignored; this avoids the overhead of transferring the value back from Lisp-Stat to Excel and is also useful if the value type cannot be automatically converted to a value meaningful to Excel.

The final step in Lines 13–15 is to call the new function with the contents of the specified columns. This uses the `CallNoValue` method of the application server.

3.3 Responding to Changes

Returning to the first setting where Lisp-Stat is the client and Excel the server, we can use

```
(setf (property xl :visible) t)
```

to make the Excel server visible and available for user interaction. At this point we might like to edit the data in the spread sheet, and it would be nice if the plot of the data could be updated automatically to reflect any changes we make. This can be arranged by registering an event listener with the `sheet` object.

An event listener for `sheet` is constructed and connected with

```
(setf sheet-listener (send event-server :new sheet))
(send sheet-listener :connect)
```

Once connected, this object receives notifications from `sheet` whenever certain events occur as a result of user interaction or executing a program. These events include changes to data and changes to the focus cell, among others. A listing of event names for Excel can be found in a reference such as [6] or using a COM browser. The event listener is now a server for which the worksheet is the client.

Notifications are ignored unless the event listener defines an appropriate method. The method name is specified by the source interface for the `Worksheet` object, and the name of the method called when the sheet's data changes is `Change`. Here is a simple definition of a `Change` method for updating the plot when the sheet's data changes:

```
(defmeth sheet-listener :change (range)
  (let ((cell (property sheet :cells 1 1)))
    (let ((data (property (property cell :CurrentRegion) :value)))
      (let ((cols (column-list data)))
        (send car-spin :clear :draw nil)
        (send car-spin :add-points (select cols '(1 3 4)) :draw nil)
        (send car-spin :adjust-to-data)))))
```

This definition ignores the `range` argument that reflects the cells that have changed and simply copies all the data from the spread sheet. A more sophisticated approach would make use of the `range` argument to reduce the amount of data transferred.

3.4 Distributed Computing

The final example illustrates the use COM in a distributed environment. A simulation is to be split over m machines. The particular simulation task used as an illustration is to evaluate the expected value of the sample median for a sample of n χ^2_d random variables using a total simulation sample size of mk with k simulations run on each machine. The simulation for each machine is handled by the simple function

```
(defun sim (n k d)
  (let ((val (make-array k)))
    (dotimes (i k (mean val))
      (setf (aref val i) (median (chisq-rand n d))))))
```

The distributed program uses a supervisor/worker model [3] in which a supervisor program starts up m worker programs on each of m machines and then collects their results. The workers are implemented by COM servers based on the prototype defined by

```
(defproto median-simulator () () auto-server)
```

The simulation on a particular machine is carried out by the `:run` method of the simulation server.

```
(defmeth median-simulator :run (n k d receiver)
  (flet ((runner (n k d rcv)
           (setf (property rcv :value) (sim n k d))))
    (async-call #'runner n k d receiver)))
```

```
1  (defproto receiver '(semaphore value) () auto-server)
2
3  (defmeth receiver :isnew ()
4    (call-next-method)
5    (setf (slot-value 'semaphore) (make-semaphore 0)))
6
7  (defmeth receiver :value ()
8    (wait-semaphore (slot-value 'semaphore))
9    (slot-value 'value))
10
11 (defmeth receiver :set-value (v)
12   (setf (slot-value 'value) v)
13   (release-semaphore (slot-value 'semaphore)))
14
15 (send receiver :add-auto-property :value :value :set-value)
```

Table 2: Value receiver object with a semaphore for synchronization

COM does not yet provide support for asynchronous calls (this will change with COM+ in Windows 2000), so the `:run` method must start the computation and then return immediately. Ideally this would be done by creating a separate thread to carry out the computation, but this is currently not possible since Lisp-Stat does not yet support multiple threads. Instead an asynchronous call mechanism provided by the `async-call` function can be used. This mechanism places the call on the event queue, where it is processed in idle time. To make the Lisp-Stat `:run` method available as a COM method, it needs to be registered by

```
(send median-simulator :add-auto-method :run :run)
```

The last argument to the `:run` method is a value receiver, a COM object owned by the supervisor that accepts the value of the computation as its `Value` property. The code for the receiver is shown in Table 2. This receiver object is also a COM server. It contains a semaphore object that is used to signal when the receiver has been given a value. The semaphore is created locked (lines 3–5). The receiver's `:set-value` method (lines 11–13) places a new value in the `value` slot and then releases the semaphore. The `:value` method (lines 7–9) waits until the semaphore is released before retrieving the contents of the `value` slot. Together these two methods make up the `Value` property of the corresponding COM object; this property is registered in line 15. This receiver is intended to be used only once; otherwise an additional lock would be needed to insure that it is not written before the value has been read.

The supervisor is implemented by the two functions shown in Table 3. The first function starts the simulations. It takes the simulation parameters and a list of machine names as arguments. For each machine the local function `start` creates a receiver object and starts a Lisp-Stat server on the specified machine (lines 3–5). The server is asked to load a file `medsim` containing the code for the simulation (line 6), and is then asked to create a new simulator (line 7). Finally, the `:run` method on this simulator is called to start its work and the receiver is returned (lines 8, 9). The function `start` is applied to each machine on the machine list and a list of the receivers is returned (line 10).

Once `start-simulations` has been called, the simulations are running in parallel on their respective machines and the supervisor needs to collect the results. This is done by the function `collect-values`. This function requests the value for each receiver and returns a list of the results. To carry out a

```
(defun start-simulations (n k d machines)
  (flet ((start (mach)
           (let ((rcv (send receiver :new))
                 (app (create-object "XlispStat.application"
                                     :server mach)))
             (invoke app :load "medsim")
             (let ((sim (invoke app :create "median-simulator")))
               (invoke sim :run n k d rcv))
             rcv)))
     (map 'list #'start machines)))

(defun collect-values (receivers)
  (flet ((collect (rcv) (send rcv :value)))
    (map 'list #'collect receivers)))
```

Table 3: Supervisor implementation

simulation using $m = 2$ machines, $k = 10,000$ simulations per machine, samples of size $n = 10$, and $d = 5$ degrees of freedom for the populations sampled, the supervisor would use an expression of the form

```
(let ((machines (list "192.168.1.3" "192.168.1.10")))
  (collect-values (start-simulations 10 10000 5 machines))))
```

This example is of course very simple, but it can be used as the basis for creating a useful framework for managing distributed computations on a network of machines supporting COM.

4 Implementation Issues

Interfacing a statistical language like Lisp-Stat to a component object system like COM requires cooperation between their respective runtime systems. This is best achieved by identifying the basic requirements and providing high-level facilities for satisfying these requirements.

The first requirement is a foreign function interface. Internal Lisp-Stat functions use a simple standard calling interface. Access to COM is implemented by C functions in a dynamic link library (DLL). Lisp-Stat provides a mechanism for declaring interfaces to arbitrary C functions that can be used to create Lisp-Stat interfaces to the C functions in the COM DLL. For example a C function with prototype

```
double fun(double x, int n);
```

is described by a Lisp-Stat declaration of the form

```
(wrap:c-function lisp-fun "fun" (:flonum :integer) :flonum)
```

Tools are provided for processing this declaration into C an intermediate function that implements `lisp-fun` with the standard internal calling convention; this function in turn calls `fun` according to its C prototype. The resulting C code is then compiled into a DLL that can be loaded and used like a standard part of Lisp-Stat.

One of the most important features of high level languages is automatic memory management, or garbage collection. For a COM interface to be usable, management of COM objects needs to be integrated with the Lisp-Stat memory management system. For example, in the expression

```
(property (property sheet :cell 1 1) :value)
```

the intermediate cell object is no longer needed once the complete expression has been evaluated. The Lisp-Stat memory management system will automatically reclaim the memory used by the Lisp-Stat representation of the COM cell object, but more is needed to insure that the COM object itself is also reclaimed. COM uses reference counts for managing its objects; the interface must therefore insure that the COM object's reference count is decremented when the Lisp-Stat representation of the object is reclaimed. This is handled by a *finalization* mechanism. The expression

(system:register-finalizer *object fun*)

insures that the function *fun* is called with *object* as its argument when *object* becomes eligible for garbage collection. When the Lisp-Stat representation of a COM object is constructed, a finalization function is registered for the representation that decrements the associated COM object's reference count when the representation is reclaimed.

Another issue is raised by event handlers. A Lisp-Stat plot object is usually released once it is closed by the user. However, if the plot is based on an Excel spread sheet and registers an event listener for events in the sheet, then the system will need to store a reference to the plot in order to execute the event handler action on the plot when events occur. This internal reference would prevent the plot from being reclaimed even after it has been dismissed by the user, thus creating a memory leak. To avoid this problem, the reference to the plot maintained by the system must be a *weak reference* that does not prevent the plot from being garbage-collected. Weak references are created with expressions of the form

(system:make-weak-box *object*)

The function `weak-box-value` called with the weak box as its argument returns *object* if it has not yet been garbage-collected, or `nil` if it has.

Finally, COM needs to be initialized before it is used, but it also needs to be uninitialized before the Lisp-Stat process exits. This can be accomplished by registering an exit function using

(system:add-exit-function *fun*)

When Lisp-Stat exits, it calls all registered exit functions in reverse order to their registration.

These four facilities provide all that is needed to allow the COM interface to Lisp-Stat to be constructed. No further access to or detailed knowledge of the Lisp-Stat internals is needed. Interfaces to CORBA or to Java [1] can also be built on these facilities. Adding similar facilities to other systems should greatly simplify interfacing them to systems like COM as well.

5 Conclusions

The Lisp-Stat/COM interface described in this paper provides a valuable tool for integrating Lisp-Stat with other applications. In addition, the experience in designing the interface has helped to identify features needed to support system integration that will be useful for other system combinations than Lisp-Stat and COM.

At a basic level, the issues in integrating COM Automation with a statistical system are similar to the issues of integrating CORBA using the dynamic CORBA facilities or Java via the Java Native Interface [5] since all three are described by a fixed set of C functions. Within COM, however, reliance on the Automation subset is somewhat limiting: There are limits on the data types that can be passed as arguments, there are efficiency limitations in particular for in-process servers, and not all clients are able to use Automation—some can only use direct binary interfaces. These direct interfaces require calls to C functions with calling conventions specific to the interface. To be able to use these interfaces, a statistical system would need to be able to generate calls to arbitrary C functions at run time. To be able to define servers that provide one of these interfaces, it would be necessary to generate arbitrary C functions at run time. Neither of these tasks can be accomplished using portable C code. Details of what needs to be done are highly dependent on both the processor architecture and the operating system. Fortunately, useful libraries supporting the generation of calls and functions are becoming available, so it may be possible to support direct COM interfaces in the future.

Another area needing further work is the use of multiple threads. Servers in particular can benefit from having multiple threads available. When they are not, the illusion of multiple threads can sometimes be created by using an event loop, and this is the approach taken by COM for single-threaded servers and clients. This approach does, however, create some subtle problems in the way it interacts with dynamic execution states such as exception handling stacks. Using multiple threads would avoid some of these issues and provide other benefits as well, and should therefore be explored.

References

[1] Ken Arnold and James Gosling. *The Java Programming Language*. Addison Wesley, 2nd edition, 1997.

[2] Guy Eddon and Henry Eddon. *Inside COM+: Base Services*. Microsoft Press, 1999.

[3] Hesham El-Rewini and Ted G. Lewis. *Distributed and Parallel Computing*. Manning, 1998.

[4] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. Calling hell from heaven and heaven from hell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, volume 34 of *ACM SIGPLAN Notices*, pages 114–125. ACM, September 1999.

[5] Sheng Liang. *The Java Native Interface*. Addison Wesley, 1999.

[6] Paul Mc Fedries. *VBA for Microsoft Office 2000*. SAMS Publishing, 1999.

[7] Microsoft COM technologies. Word Wide Web. `www.microsoft.com/com`.

[8] Object management group home page. World Wide Web. `www.omg.org`.

[9] Alan Pope. *The CORBA Reference Guide*. Addison Wesley, 1998.

[10] Randal L. Schwartz, Erik Olson, and Tom Christiansen. *Learning Perl on Win32 Systems*. O'Reilly & Associates, 1997.

[11] Ellen Siever, Stephen Spainhour, and Nathan Patwardhan. *PERL in a Nutshell*. O'Reilly & Associates, 1999.

[12] Luke Tierney. *LISP-STAT: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics*. J. Wiley & Sons, New York, NY, 1990.

[13] Aaron Watters, Guido van Rossum, and James C. Ahlstrom. *Internet Programming with Python*. M&T Books, 1996.

[14] Brent B. Welch. *Practical Programming in Tcl and Tk*. Prentice-Hall, Upper Saddle River, NJ, 3rd edition, 1999.