Astronomical Data Analysis Software and Systems IX
ASP Conference Series, Vol. 216, 2000
N. Manset, C. Veillet, and D. Crabtree, eds.

Event Driven Programming with Glish

D. R. Schiebel

National Radio Astronomy Observatory[1], *520 Edgemont Road, Charlottesville, VA 22903-2475*

Abstract. Glish is a powerful whole-array scripting language with Perl regular expressions. Its most distinctive feature, however, is its asynchronous event handling mechanism. This is also the mechanism by which Glish is extended. Separate processes (called "clients") are developed using Glish's C++ library. Once developed they are seamlessly integrated into scripts. The result is a distributed, loosely coupled system made up of clients woven together by a script which controls and directs the flow of events between clients. Systems developed using Glish are very modular and are easily adapted to changing requirements.

1. Introduction

The functionality in Glish applications is divided between Glish scripts and specialized clients. The Glish scripting language is powerful, but it is much slower than compiled code. For compute-intensive portions of the application, specialized clients are created. These clients are regular Unix processes which receive and generate events using the Glish client library. Once these clients are created, their functionality can easily be used by Glish scripts. As the rest of this paper shows, this hybrid approach to application development is very powerful allowing the programmer to trade off scripting against compiled C++ code as an application matures.

2. History

Glish was initially developed to address the data collection needs of collider experiments. It started in 1988 as a collection of libraries, and later that year a simple scripting language was added when it was used for data collection on the Fermilab Tevatron. By 1991 most of the funding for Glish was coming from the Superconducting Super Collider, where it was used in the Magnet Test Facility in 1992 and considered for use in the SSC control system. During this time Glish went through two rewrites to arrive at its current syntax and semantics (Paxson & Saltmarsh 1993).

[1]The National Radio Astronomy Observatory is a facility of the National Science Foundation operated under cooperative agreement by Associated Universities, Inc.

Most of the funding for Glish dried up with the demise of the SSC in 1993. While it did see some limited use with the Advanced Light Source at Lawrence Berkeley National Laboratory and the Relativistic Heavy Ion Collider at Brookhaven National Laboratory, further development was limited.

During this same period, however, the AIPS++ project was doing initial design work. At that point, AIPS++ had developed several library modules but lacked a command line interface (CLI) and an interprocess communications (IPC) layer. The IPC layer was necessary to allow systems using AIPS++ to be distributed across a network taking advantage of services like compute servers. The CLI had to allow binding of AIPS++ functionality as it was developed, but this had to be done without encumbering the CLI by requiring changes to the parser each time new functionality was added to AIPS++. Glish was a nice fit for these requirements, and it was adopted by AIPS++ in 1994. Since that time Glish has continued to be developed by the AIPS++ group.

3. Language Features

While the event handling features are the key technology in Glish, the scripting language is also powerful. It allows users to perform much of their data analysis directly in Glish. The syntax and semantics are loosely based upon the "S" statistical analysis language developed at ATT (Becker, Chambers, & Wilks 1988).

3.1. Dynamic Typing

Among the first things that users of statically typed languages like C and Fortran will notice about Glish is the fact that it is dynamically typed. The type of variables can change as they are used. There are eight basic types – numeric, string, regular expression, record, function, file, fail, and agent. Numeric values can be freely combined, and the result is promoted to the type with the highest precedence. For example (the Glish prompt, `-` or `+` when more input is expected, is included in many examples):

```
- x := 2
- x := x + 3+2i
- print x
5+2i
- sin(x + 'Hello World')
<fail>: non-numeric operand in expression: (x + Hello World)
        Stack:  sin()
```

Here the type of `x` goes from integer to complex, but combining numeric and non-numeric values results in a fail value. Fail values are the way errors are handled in Glish. These values are automatically propagated; any use of a fail value generates another fail value. In the case above, the sine function is not invoked.

For numeric types, Glish supports whole array arithmetic operations:

```
- x := 2:10
- x * 2
```

```
[4 6 8 10 12 14 16 18 20]
```

Multidimensional arrays are also supported:

```
- y := array(1:24,3,4,2)
- print y * y
[[1:3,,1]
    1 16 49 100
    4 25 64 121
    9 36 81 144
[1:3,,2]
    169 256 361 484
    196 289 400 529
    225 324 441 576]
```

but operations are still array oriented rather than vector oriented, i.e. the operation is applied to corresponding elements rather than by rows or columns.

Aside from typical array indexing, there are three different possibilities for element selection – pick, mask, and slice. The latter two are used most often. A mask is the simplest; it is just a boolean array which acts as a bit mask for the array:

```
- y[ y % 2 != 0 ] := 0
- print y
[[1:3,,1]
    0 4 0 10
    2 0 8 0
    0 6 0 12
[1:3,,2]
    0  16 0  22
    14 0  20 0
    0  18 0  24]
```

which will set all of the odd elements in the array to zero. Slices are used to select contiguous pieces of the array:

```
- print y[1:2,1,]
[[1:2,]
    0 0
    2 14]
```

In this case, the first two elements from the first column of each plane was selected.

3.2. String Type

Glish has rich string manipulation due to the inclusion of Perl's regular expressions. So it is easy to match and manipulate strings:

```
- 'hello world' ~ s/[aeiou]//g
hll wrld
- 'hello world' ~ m/[aeiou]/g
3
```

Glish's regular expression library is taken directly from Perl so it includes nearly the full set of Perl 5.0 (Wall, Christiansen & Schwartz 1996) regular expression capabilities. The main difference is that the tilde (~) operator applies a regular expression to a string without modifying the string. Like Perl, the =~ operator modifies the string itself when doing substitution. Another difference is that in Perl there is no distinction between a regular expression and the application of the regular expression. In Glish, the regular expression is like any other value; it can be passed as parameters etc. and applied with ~, =~, or !~ when needed.

3.3. Records

Record allow users to store heterogeneous collections of data:

```
- r := [offset=13, time='Thu Aug 26 18:11:05 UTC 1999', iter=25]
- r.iter +:= 1
- print r["time iter"]
[time=Thu Aug 26 18:11:05 UTC 1999, iter=26]
```

As this example shows, records can be addressed either with the dot notation common to records in "C" or array notation. Records can also contain other records:

```
- points := [=]
- points[len(points)+1] := r
```

Using array addressing, numbers representing the offset into the record can be used to access elements, as above. Records can be nested arbitrarily deep.

3.4. Control Structures

Glish supports all of the standard programmatic control structures, i.e. if/else, while, for, next, break. Each of these constructs are generally similar to the "C" version except that the `for` loop is vector oriented (as with the Bourne shell):

```
- for (i in 1:3) {
-     print sin(i)
- }
```

here `i` iterates over the vector [1, 2, 3].

3.5. Functions

Glish supports functions:

```
func sort(x) {
  if ( len(x) <= 1 ) return x
  pivot := x[random(1,len(x))]
  return [sort(x[x < pivot]), x[x == pivot], sort(x[x > pivot])]
}
```

Here vector masking is used to implement a quick sort function. The vector is partitioned and then the function is used to recursively sort those elements greater than or less than the pivot.

Function arguments can have default values, and arguments are matched by position or by name. So for example, if we have a function:

```
func frame ( parent=F, relief='flat', padx=0, pady=0 ) { }
```

It can be called like `frame( T, padx=10 )` which would set `parent` to true (`T`) and `padx` to `10` while accepting the defaults for the remaining arguments.

3.6. Event Handling

The aspects of Glish covered so far are available in other languages. The event handling mechanism in Glish is relatively unique. It allows users to create client processes which can communicate with the Glish interpreter by sending and receiving events. An event is simply a name plus a value. The value can be any valid Glish value, except functions, agents, or regular expressions.

Events can be either synchronous or asynchronous. Here are some examples of synchronous events:

```
- x := seq(0,10,.001)
- y := sin((x+2)*2)+sin((x+1.85)*3)+sin((x+0.25)*5)
- fft := client('fft_client')
- f1 := fft->realtocomplex(y)
```

The results of events sent to a client, here `fft`, can be used in expressions. In this case, it is just assigned to a variable. A script can also explicitly wait on a result from a client:

```
- fft->realtocomplex(y)
- await fft->*
- f1 := $value
```

This is basically equivalent to the `f1` assignment above, but here an event is sent to the FFT server and then the script waits for the client to send an event in response.

Events can also be handled asynchronously. This is done by setting up `whenever` blocks which will perform some action when particular events arrive:

```
- whenever fft->result do {
+     plotter->clear()
+     plotter->line( $value.x, normalize($value.y) )
+ }
```

This will automatically send the results of the FFT to a plotter after performing any normalization.

While these examples are very trivial, the concept is very powerful. Distilling the interface to clients down to a basic event dialog results in a minimal well defined interface which makes it easy to plug new clients into the system to replace old ones. Events from clients generally pass through the Glish interpreter so any necessary manipulation can be done in the controlling Glish script before passing the result off to another client. The event oriented nature of Glish is consistent with a variety of environments ranging from GUI design to control systems.

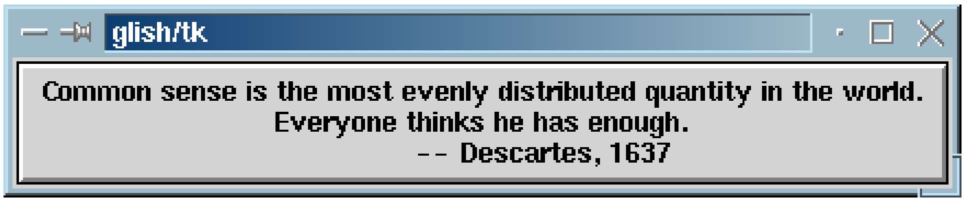

Figure 1. Simple GUI.

3.7. Glish/Tk

Currently the best example of binding a client process's functionality to Glish is the glishtk client. This client makes the basic Tk (Welch 1997) widgets available to Glish scripts. It does this by making each widget look like a client which accepts and generates events.

```
- f := frame()
- b := button(f,text=paste(shell('fortune'),sep='\n'))
- whenever b->press do
+     b->text(paste(shell('fortune'),sep='\n'))
```

This example (see Figure 1) updates a button's text with output from fortune each time the button is pressed. The Tk widgets reside in a client process but are manipulated from the Glish script. When a widget is created in the client, an associated client descriptor is created in Glish. When the user sends events to this descriptor, the effect is applied to the associated widget in the client. As this example shows, client descriptors can be passed back to the client. In this case, the frame (`f`) is passed to the client when creating the button (`b`). This capability is similar to that of object request brokers.

4. Applications

4.1. Parkes Multibeam

Glish and AIPS++ are used to collect and calibrate the data generated by the thirteen-beam Parkes[2] 21 cm Multibeam Receiver (Barnes 1998). The realtime calibration system consists of a few Glish clients and several Glish scripts (see Figure 2). The Multibeam correlator generates RPFITS (Radio-physics FITS) files. The correlator currently has a 5 second cycle time, and two polarizations per beam are written at the end of each cycle. So each cycle 26 spectra each with 1024 channels are written in RPFITS format. So the data rate is about 1.2 MB per minute. It is at this point that the Glish-based reduction package accesses the data.

A RPFITS Glish client (`RPFITS Reader`) reads the correlator data and creates a Glish record which represents the data generated in one cycle of the correlator. This record is then posted to Glish. At this point, the Glish script

[2] `http://www.parkes.atnf.csiro.au/`

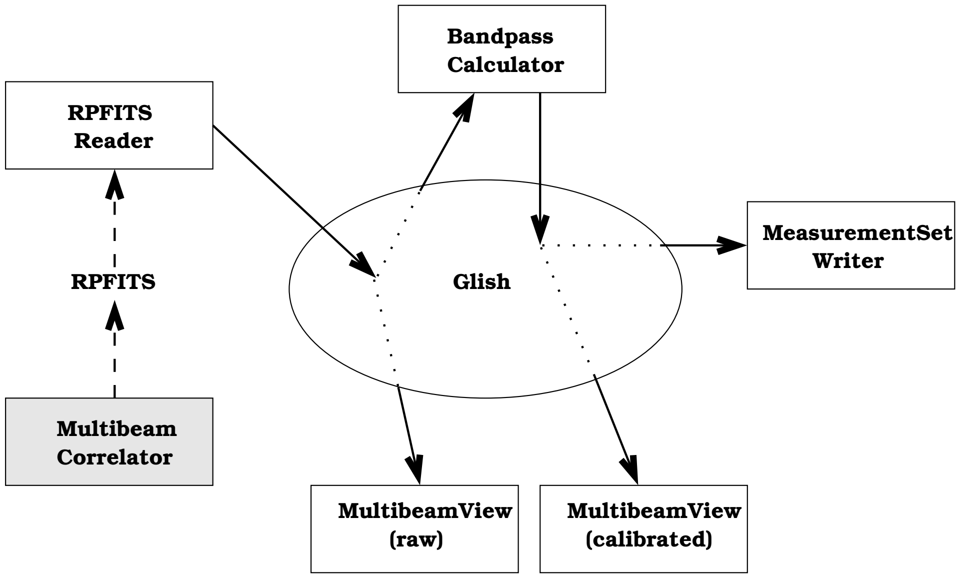

Figure 2. Parkes MultiBeam System Diagram.

directs the record to the `Bandpass Calculator`, and if monitoring of the raw data is enabled, the record is also directed to the `MultibeamView` client for plotting.

The `Bandpass Calculator` is the most sophisticated client in the system. It does robust removal of gross bandpass from the spectra and calibration of the spectra using the system temperature values from the correlator in realtime. The bandpass is removed based on a statistical estimate of the bandpass at the position and time of a particular spectrum. The estimate is based on a two minute window of data where earlier and later spectra observed by the same beam are used to arrive at a statistical estimate of the bandpass. This client can also do robust zeroth order baseline removal, spectral smoothing and velocity frame conversion. The corrected spectra are then posted to Glish, and the script sends the event to the `MeasurementSet Writer` client for storage, and to the `MultibeamView` client for realtime display. The system is controlled with GUIs built using Glish/Tk.

The key feature of Glish systems such as this is that capabilities are bundled up in clients which are encapsulated based on events which they accept and generate. This level of encapsulation ensures that it is easy to substitute new clients for old ones without disturbing the rest of the system. The events which the clients generate flow through the Glish interpreter so that it is very easy to adjust the behavior of the system with changes to the controlling scripts. This sort of flexibility is especially valuable during prototyping. In fact, many calculations can be implemented in the Glish scripting language during the prototyping period, and later those which prove to be too slow can be moved to compiled C++ code in a client.

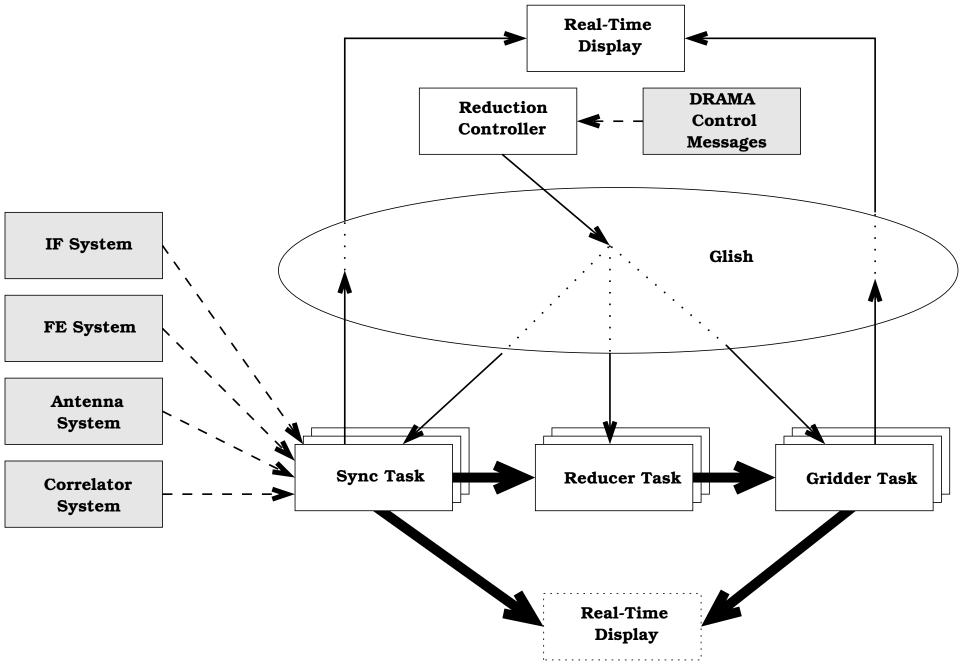

Figure 3. ACSIS System Diagram.

4.2. JCMT ACSIS

The reduction system for the Auto-Correlation Spectrometer Imaging System[3] (ACSIS) for the James Clerk Maxwell Telescope (JCMT) is also based on Glish. It has goals similar to that of the Multibeam system except that the data rate is greater by more than an order of magnitude; the maximum data rate is 10 MB per second.

DRAMA[4] is used to control the reduction system, and it is used to acquire the data to be reduced (see Figure 3). The processing load is divided into eight parallel reduction pipelines. The data is collected by the `Sync Task` which receives data from the telescope systems using DRAMA and then organizes the data into chunks for use by the reduction system. The data is then posted as a Glish value destined for the `Reducer Task` and the `Real-Time Display`.

This data event would normally flow through the Glish interpreter before being delivered to the target clients. However, because the data rate is so high direct pipes are established between the key clients (represented as bold arrows in Figure 3). These direct pipes are established using the `link` command from the controlling Glish script. This command tells a client to direct specific events directly to another client and optionally rename the event in the process. These links can be created and removed dynamically from the Glish script. Nothing changes from the client's perspective whether the event is being delivered

[3] `http://www.drao.nrc.ca/science/jcmt_correlator/acsis/pdr/pdr03.pdf`

[4] `http://www.aao.gov.au/drama/html/dramaintro.html`

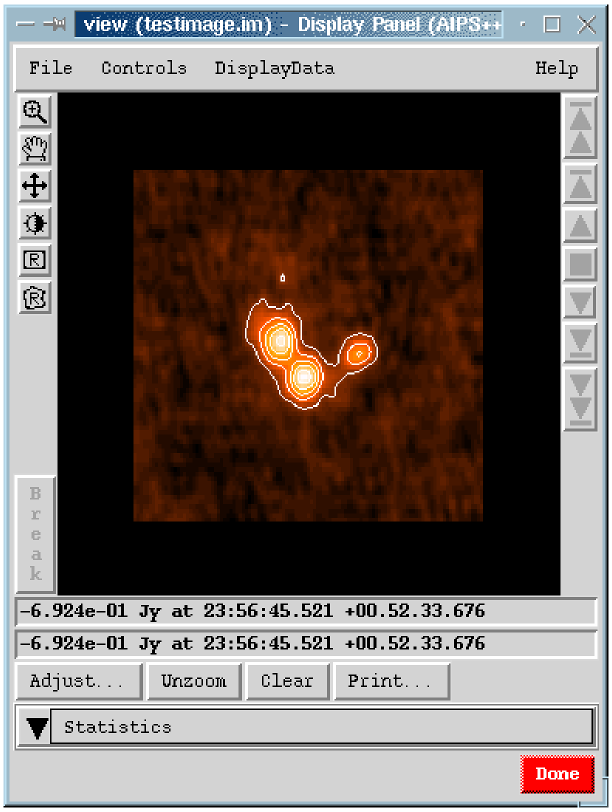

Figure 4. AIPS++ Viewer.

through Glish or directly to another client. This is true for the sender and the receiver; the link mechanics are handled by the Glish library.

Reduction recipes control the operation of the `Reducer Task` (Lightfoot et al. 2000). The output flows to the `Gridder Task` which is based on the gridder used in the Multibeam project (see § 4.1.). It creates a three dimensional data cube and writes it to disk.

4.3. AIPS++

AIPS++[5] (Astronomical Information Processing System) is a software system designed for calibration, editing, and imaging of data collected from radio telescopes. Glish is the glue which binds the pieces of AIPS++ together. Not only does it supply the transport layer which AIPS++ components use to communicate, but it also supplies the scripting language which allows users to direct and combine components in novel ways.

The AIPS++ system is composed of a number of clients each supplying a group of related capabilities. This sort of functionality decomposition gives the user the option of either using an existing client or developing a new specialized

[5] `http://www.nrao.edu/aips++/aips++.html`

client using the AIPS++ libraries. The typical AIPS++ user will simply use clients which already exist. In most cases, the user won't even know that many processes are cooperating to manipulate the data. All of these details are hidden from the user by functions.

Figure 4 shows the image viewer which is one of the end-user applications in AIPS++. These applications are very useful, but they are actually veneer atop a toolkit. It is the toolkit which makes AIPS++ powerful. The toolkit is made up of encapsulated components which makes it easy to customize and combine the components. Just as C++ encourages encapsulation at the source code level, Glish encourages encapsulation at the component level. Users can select the components they wish to use and replace those which do not fit their needs. In Figure 4, the imager component is used to produce the image, and the viewer component is used to display it. The viewer component is only the square canvas on which the image is drawn. All of the other widgets drawn around the viewer are Glish/Tk widgets. Therefore, it is easy for a user to take the viewer, but control it with application specific widgets. Non-graphical components can be customized or replaced in a similar manner.

5. Conclusion

Glish is a flexible general purpose scripting language, but it is also well suited for use in control systems. It allows users to construct and easily modify loosely coupled distributed systems. The addition of AIPS++ tailors Glish for astronomy by providing all of the scripts and clients necessary to reduce astronomical data. Because the Glish scripting language is powerful, users can do much experimentation in Glish, moving portions to a C++ client when they prove to be too slow.

References

Barnes, D. G. 1998, in ASP Conf. Ser., Vol. 145, Astronomical Data Analysis Software and Systems VII, ed. R. Albrecht, R. N. Hook, & H. A. Bushouse (San Francisco: ASP), 32

Becker, R. A., Chambers, J. M., Wilks, A. R. 1988, The New S Language (Pacific Grove, Wadsworth & Brooks)

Lightfoot J., Dent W., Willis, A. G., Hovey, G. 2000, this volume, 502

Paxson, V., Saltmarsh, C. 1993, Proceedings of the Winter Technical Conference (Berkley: USENIX Assoc), 141

Wall, L., Christiansen, T., Schwartz, R. 1996, Programming Perl 2nd Edition (O'Reilly & Associates)

Welch, B. B. 1997, Practical Programming in TCL and Tk 2nd Edition, (Prentice Hall PTR)