A Language, Server, and C++ Class Library for Event Sequencing

V. Paxson[1]
Computer Science Department
Cornell University, Ithaca, NY 14853

C. Saltmarsh and M. Allen
Superconducting Super Collider Laboratory
Dallas, TX 75237

M. Kane
Lawrence Berkeley Laboratory
Berkeley, CA 94720

Abstract

Many accelerator control tasks are well specified as a sequence of operations, each step of which is contingent upon the status of other operations or sequences. To enable the rapid creation and modification of such sequences we have devised the *Glish* language. Sequences are described in *Glish* in terms of asynchronous hardware and software *events*, where an event is the notification that a particular thing has happened and the value associated with that notification. Events are produced and consumed by Unix processes. One uses *Glish* to write expressions which specify how events associated with the processes depend on combinations of other events. We describe the *Glish* server, which manages *Glish* events distributed across a network, a C++ class library for *Glish* clients, and operational experience gained in using *Glish* to control an accelerator physics experiment on the Fermilab Tevatron.

[1]Work done at Lawrence Berkeley Laboratory, Berkeley, CA, 94720, and supported in part by the United States Department of Energy under Contract Number DE-AC03-76SF00098.

Introduction

One problem when dealing with large systems is that of *integration*: how to connect together different components of the system in order to effectively use the system as a whole to accomplish some task. Often these tasks have natural descriptions as a *sequence* of operations carried out by components of the system, in which each step in the sequence depends on the status of other operations or sequences.

One such large system is the software system for controlling an accelerator. The system's components are the programs which carry out particular functions such as setting power supplies or computing physical properties of the beam. There are many such programs, and sequencing the interactions of the programs can prove both difficult and unwieldly. The programs often have subtle interdependencies which change with the task at hand, and often the way they are sequenced must be changed in the middle of the task as new problems arise. What's needed, then, is a way of describing how to sequence the programs needed by a particular task in such a way that:

- creating descriptions is simple and rapid;
- given a description, the necessary programs can then be automatically sequenced;
- when describing a new task one can easily use the already-existing descriptions of other, sub-tasks;
- descriptions are *extensible*; that is, they are easy to modify when unforeseen requirements or changes crop up;
- the descriptions work in a distributed environment; that is, one in which the various programs can run on different hosts connected by a network.

We have developed a prototype of a system for specifying and executing such sequences of operations. The cornerstone of the system is the *Glish* language, which is used to specify sequences in terms of the programs they require and the asynchronous hardware and software *events* generated by those programs. *Glish* sequences are interpreted by a distributed *sequencer*, which manages the activation of the required programs and the flow of events between them. The *Glish* system also includes a group of utility programs and a C++ class library to aid in integrating new programs into the system, and a graphical "Control Panel" program for monitoring and controlling the progress of sequences.

The Glish System

The basic model of the *Glish* system is that the various control programs are written in terms of *events* which they generate and respond to. An event is notification that something has happened plus an associated value. For example, a program which uploads beam position monitor (BPM) readings from a piece of hardware might generate a `new-readings-available` event whenever it has a new set of readings, and the value of the event could be the name of the database table where it stored the readings. The program might respond to `change-sensitivity` events, which would tell it to adjust the range of the raw BPM readings by an amount equal to the value of the event. In *Glish* lingo, event-driven control programs such as these are called *clients*.

The central clearing house where all events are gathered and re-routed is a distributed program called the *sequencer*. To define how to sequence a control task, the user writes a description of the sequence in the *Glish* language. The description includes the clients or other sequences which the sequence requires and a series of "event expressions" of the form: "whenever these events occur, create a new event with the following value." The hope behind our prototype is that the *Glish* language will prove rich enough to describe a large spectrum of high-level control tasks, from computing tunes and chromaticities to tweaking up the vacuum system, or coordinating off-line analysis, or improving beam emittances, or recovering from a quench.

The sequencer itself responds to events sent to it. To start up a new sequence, an event is to the sequencer telling it to process the file containing the description of the sequence. The sequencer then makes sure that all the required clients and sub-sequences are running, and begins creating and routing events as directed by the sequence's description.

It's probably simplest to explain the details of the *Glish* system using an extended example.

A Simple Sequence

Suppose we wish to take BPM readings each cycle and FFT them to find the corresponding tune. We have a CAMAC crate which digitizes the raw BPM readings and stores them in its internal memory.

We need two programs. The first, which we'll call `read-bpms`, knows how to talk to the CAMAC crate. It uploads monitor readings from the crate into a shared memory region called "readings". When it's done, it generates an event `got-shot` whose value is the name of the shared memory region. When the second program, `fft`, receives a `compute` event, it attaches to the shared memory region indicated by the value of the event, and computes the FFT. It then displays the X and Y tunes on the operator console.

This sequence would be written as:

```
sequence get-tune:
    requires
        read-bpms, fft;

    read-bpms.got-shot --compute-> fft;
```

The `requires` line specifies which clients are needed by the sequence. The sequencer will make sure that each client is running when needed.[2]

The final line in the sequence above, called an "event expression", is the actual sequencing operation. It is read, "whenever `read-bpms` generates a `got-shot` event, send a `compute` event with the same value to `fft`." If we wanted to change the value of the `compute` event to "old-readings", we could have done so using[3]:

```
    read-bpms.got-shot --compute-> fft ["old-readings"];
```

Distributed Sequences

Suppose now that we want to run `read-bpms` on a remote computer called `camac-sun` but would like to run `fft` locally. To do so, the region of shared

[2]`requires` can also be used to rename clients. For example, if the BPM-reading program was called something fairly uninformative, such as `lecroytweak`, it could be renamed using "`requires read-bpms = lecroytweak, fft;`" and now would be referred to using the name `read-bpms`.

[3]The syntax here is admittedly awkward.

memory must be transferred to the local machine[4]. We need another program, which we'll call `get-shm`. When `get-shm` receives a `go` event, it interprets the value of the event as two arguments: the first is the name of the remote machine to fetch from, the second the name of the shared memory region on that machine. `get-shm` generates a `ready` event when it has finished its task; the value of the `ready` is the name of the local shared memory region, which may be different from the name of the remote region.

The sequence now looks like:

```
sequence get-tune:
    requires
        read-bpms @ camac-sun,
        get-shm, fft;

    read-bpms.got-shot --go-> get-shm ["camac-sun" "readings"];
    get-shm.ready --compute-> fft;
```

The last two lines are read: "Whenever `read-bpms` generates a `got-shot` event, send a `go` event to `get-shm` with values 'camac-sun' and 'readings'. Whenever `get-shm` generates a `ready` event, send a `compute` event to `fft`."

Because of the `@ camac-sun` in the `requires` part of the sequence, `read-bpms` will now run on the remote host. This change is transparent to `read-bpms`; its interface to the sequencer is simply that when it makes a Receive request it gets its next event and when it makes a Send request it sends out any event it desires. It has no idea where the events come from or are going to; that information is fully isolated inside the sequencer.

Default Event Names

One design goal in developing *Glish* was that it be as simple as possible to write down most sequences. To this end, several syntactic shortcuts are used. Two of these are that a missing `.event` name defaults to `ready` and that the operator `->` is equivalent to `--go->`. So the above pair of event expressions can be written:

```
    read-bpms.got-shot -> get-shm ["camac-sun" "readings"];
    get-shm --compute-> fft;
```

[4]Ideally we'd like the *Glish* system to do this for us automatically.

Furthermore, a group of `--name->` expressions can be strung together. For example, we could also write the above pair of expressions as[5]:

```
read-bpms.got-shot ->
    get-shm ["camac-sun" "readings"] --compute->
        fft;
```

Given the defaults of `ready` and `go` for events received and sent out by the sequencer, it becomes natural to write clients such that these events describe their most common actions. For example, since `read-bpms`' usual action is to generate `got-shot`, it might instead simply generate `ready`. Similarly, `fft` might do its computation any time it is sent a `go`.

Referring to Previous Event Values

In further developing the sequence, we find that it's dissatisfying that `get-shm` be given the name of the shared memory region explicitly. If `read-bpms` ever changed the region's name, we'd have to edit the sequence to reflect the change, or, worse yet, would forget to do so and the sequence would no longer work. To fix this problem, the sequencer remembers the most recent value of every event it receives. Then when associating values with a new event, one can refer to the value of any previous event. So we could write the event expression as:

```
read-bpms.got-shot ->
    get-shm ["camac-sun" read-bpms.got-shot] --compute->
        fft;
```

Two General-Purpose Glish Clients

Suppose now that we want to archive each shot we take to tape, along with an associated shot number and timestamp. We need a `tape_archive` client, which, upon receiving a `go` event, interprets the values of the event as the shot number and timestamp to put in the archive header and the name of the shared memory region to write out on tape.

[5]Newlines are ignored in event-expressions, which are terminated with explicit semi-colons.

We also need a way to generate shot numbers and timestamps. Rather than building these into `read-bpms`, we'd like them to be separate clients to keep `read-bpms` from becoming too heavy with these sorts of features.

The *Glish* system comes with two clients to help us. The first, called `counter`, is used for counting up (or down) an interval. `counter` is configured by sending it a `go` event with up to three values, which are interpreted as the starting point, ending point, and step, respectively. Each time `counter` receives a `count` event, it generates a `ready` event with the current value of the counter and then increments the counter toward the ending point. When it goes beyond the ending point it generates a `done` event. All numeric values can be either integers or floating-point.

The second helpful client is `std_client`, which is used to interface non-*Glish* programs into the *Glish* system. When `std_client` receives a `go` event it interprets the associated values as the name of a program to execute and its arguments. It then creates two Unix pipes to the program and hooks them up such that whatever it writes into the first pipe appears as the program's standard input, and whatever the program writes to its standard output appears to `std_client` in the second pipe. Then whenever `std_client` receives an event other than `go`, it writes the event's value into the program's standard input pipe. Whenever the program writes a string to its standard output, that value is then turned into the value of a `ready` event.

Here, then, is the above sequence expanded to keep track of shot numbers and timestamps and to do archiving.

```
sequence get-tune:
    requires
        read-bpms @ camac-sun,
        get-shm, fft, counter, std_client, tape_archive;

    # start shot counter at 1
    get-tune.activate -> counter ["1"];

    read-bpms.got-shot ->
        get-shm ["camac-sun" read-bpms.got-shot] --compute->
            fft;

    read-bpms.got-shot --count-> counter; # increment shot counter
```

```
read-bpms.got-shot -> std_client ["date"];

(counter.ready & std_client.ready) ->
    tape_archive
        [counter.ready std_client.ready get-shm.ready];
```

To use our counter we must first initialize it. Whenever a sequence is first activated, it generates an `activate` event (similarly, whenever a client exits it generates a `terminate` event). The event expression above will then be triggered and send `counter` a `go` to configure it to start at 1. As no upper bound or step is given, it will count upwards indefinitely, advancing each time by 1.

Now whenever `read-bpms` generates a `got-shot`, a `count` event will be sent to the counter and a `go` event to `std_client`, telling it to fire up the Unix `date` tool, which will write the current date to its standard output (which in turn will become a `std_client.ready` event). Whenever both `counter.ready` and `std_client.ready` have happened, a `go` will be sent to `tape_archive`. The values of the `go` will be the values of the last `get-shm`, `counter`, and `std_client ready` events. As per the shorthand mentioned above, since the `.name` operator defaults to `.ready`, the last event expression could have been written:

```
(counter & std_client) ->
    tape_archive [counter std_client get-shm];
```

Something worth noting is that once the `&` operator *fires*, that is, once both of its events have happen, it then resets, and will require *both* events to again happen for it to fire again. In the above example, if a `counter.ready` event is followed by 10 `std_client.ready` events, the `&` will fire only once. If then another `counter.ready` event occurs, then the `&` will fire once more, since it will then have seen both a `counter.ready` and a `std_client.ready` event since it last fired.

More Glish Event Operators

There are a number of other event operators similar to `&`:

- "`event1 | event2`" fires whenever *either* event1 or event2 happens.

- "`event1 &! event2`" fires whenever event1 happens and event2 has not happened since the last time the `&!` fired.

- "`event1 &~ event2`" fires whenever event1 happens and event2 has *failed*. Failure is defined as "the sequencer tried to run the client(s) corresponding to the event and they have not yet acknowledged to the sequencer that they are up and running".[6]

- "`event1, event2`" allows events to be grouped. The operator fires whenever the right-hand event fires. It's usually used to send an event to a group of clients. For example,

  ```
  operator.check-point --archive->
      read-bpms, fft, counter;
  ```

  means that whenever the `operator` client generates a `check-point` event, send an `archive` event to `read-bpms`, `fft`, and `counter`. As usual, the value of the event would be the value of the `check-point` event, and, as one might expect, the value can be changed for particular programs:

  ```
  operator.check-point --archive->
      read-bpms ["previous"], fft, counter;
  ```

  will behave as the previous expression except that the value of the `archive` event sent to `read-bpms` will be "previous". That value could also be sent to all three of the programs using:

  ```
  operator.check-point --archive->
      (read-bpms, fft, counter) ["previous"];
  ```

  The parentheses are needed because the "`[]`" operator has higher precedence than the "`,`" operator.

[6] This operator has not yet been implemented.

The event operators are all left-associative, and their precedence, from highest to lowest, is: `"."`, `","`, `"&"`, `"|"`, `","`, `"->"`. `"&!"` and `"&~"` have the same precedence as `&`. Parentheses override precedence. We find that when parentheses are not strictly called for using them still often helps to clarify precedence.

Another note regarding operators: one may not think of `"."` as an operator in the sense of the others, but it is. We can write

```
(read-bpms.error | fft.error) ->
    alarm ["Error computing tune!"];
```

as

```
(read-bpms | fft).error ->
    alarm ["Error computing tune!"];
```

Subsequences

In the above example we added time-stamping and archiving to the "get tune" sequence. This is a bit distasteful as the extra steps have little to do with the idea of "get tune" in particular. Their presence muddies the sequence's modularity, and therefore decreases the likelihood that the sequence can be easily reused.

Suppose that the `fft` program in the example above generates a `ready` event whose value is the X and Y tunes, rather than displaying the tunes directly. Then we can write a stripped-down `get-tune` sequence:

```
# Computes X and Y tunes every time a new shot comes in.
# The tune values are reported in a .ready event and the
# name of the local shared memory region with the current
# shot is reported in a .region-name event
sequence get-tune:
    requires
        read-bpms @ camac-sun,
        get-shm, fft;

    read-bpms.got-shot ->
        get-shm ["camac-sun" read-bpms.got-shot] --compute->
```

```
        fft --ready-> get-tune;

    get-shm --region-name-> get-tune;
```

The last line reads, "whenever `get-shm` generates a `ready` event, send a `region-name` event to `get-tune` with the same value, i.e., the name of the shared-memory region." We could now write a "get tunes and archive them" sequence, which looks like:

```
# computes X and Y tunes for each shot that comes in, archiving
# them to tape along with a shot number and a timestamp
sequence archive-tune:
    requires
        sequence get-tune,
        counter, std_client, tape_archive;

    # start shot counter at 1
    archive-tune.activate -> counter ["1"];

    get-tune --count-> counter;
    get-tune -> std_client ["date"];
    get-tune --display-> console ["Tune values are:" get-tune];

    (counter & std_client) ->
        tape_archive [counter std_client get-tune.region-name];
```

To run the `archive-tune` sequence requires the `get-tune` sequence. As far as `archive-tune` is concerned, `get-tune` is indistinguishable from any other program which receives and generates events[7]. Now whenever `get-tune` generates a `ready` (and that happens as soon as it *receives* a `ready`, i.e., whenever `fft` generates a `ready`), the counter will be told to count, the date program will fire up, and a `display` event will be sent to the `console` program, which presumably displays a message on the operator console whenever it receives a `display` event. Furthermore, when both the new shot number and date are ready, the tape-archiver will archive the shared memory region reported by `get-tune`'s `region-name` event.

[7]Save for the `sequence` keyword in the `requires` list, which perhaps should be done away with.

`archive-tune` cannot look inside `get-tune`'s inner workings and access those events directly[8]. That is, it could not, for example, increment the counter and create a timestamp whenever `read-bpms` generates a `got-shot` event. It can only see `get-tune`'s events. But one could easily add to the `get-tune` sequence the line

```
read-bpms.got-shot --got-shot-> get-tune;
```

to make the event accessible to `archive-tune` if needed (it would then be known as `get-tune.got-shot`).

Parameterizing Sequences

Suppose that before `read-bpms` is used it needs to be configured with the name of the CAMAC crate it is supposed to talk to and with a file of settings to download into the crate. It would be natural to make it accept `go` events specifying those values. Furthermore, let's suppose that any time `get-tune` receives a `reset` event we want to reconfigure `read-bpms`. Then we would add something like the following to `get-tune`:

```
get-tune.reset ->
    read-bpms ["CELL4" "injection-settings"];
```

Actually, though, we would like to be able to easily change the configuration, as this lets us make `get-tune` more general purpose. To do so, we can write the sequence as:

```
sequence get-tune [camac-crate camac-settings]:
    requires
        read-bpms @ camac-sun,
        get-shm, fft;

    read-bpms.got-shot ->
        get-shm ["camac-sun" read-bpms.got-shot] --compute->
            fft --ready-> get-tune;

    get-shm --region-name-> get-tune;
```

[8]Actually, as *Glish* is currently implemented, it can; but this is a misfeature.

```
get-tune.reset -> read-bpms [camac-crate camac-settings];
```

The parameterization works as follows. Whenever `get-tune` receives a `go` event, the values of the event are bound to the variables `camac-crate` and `camac-settings`. These names can then be referred to throughout the body of `get-tune`. In this case, whenever `get-tune` receives a `reset` it will reconfigure `read-bpms` with the latest configuration values. We could add to `archive-tune`

```
archive-tune.activate ->
    get-tune ["CELL4" "injection-settings"];
archive-tune.activate --reset-> get-tune;
```

to initially configure the CAMAC crate as was done above. We could then add other configuration lines to `archive-tune`, such as:

```
archive-tune.at-flat-top ->
    get-tune ["CELL4" "flat-top-settings"];
```

This expression would then change the CAMAC settings to those appropriate for flat-top whenever `archive-tune` is told that we're now at flat-top.

Relational Operators

In addition to the event operators mentioned above, glish supports relational operators for comparing the values of events. For example, if in `archive-tune` we want reset the shot counter after every 10th shot, we could add the line:

```
counter == "10" -> counter ["1"];
```

Then whenever a `counter.ready` event has the value `10`, a `go` event would be sent to `counter` telling it to start at 1 again. Comparisons can be made with either strings or numbers. Strings can be tested for equality or inequality, such as:

```
get-tune.tune-type == "horizontal" ->
    read-bpms ["HCELL4" "injection-settings"];
get-tune.tune-type != "horizontal" ->
    read-bpms ["VCELL4" "injection-settings"];
```

Numbers can be integers or floating point. If a number is compared with a string the string is first converted to a number if possible, so the above example with `counter` could have been written without quotes around the `10`. Numbers can also be compared for inequality:

```
(get-tune >= 9.18 & get-tune <= 9.21) ->
    alarm ["Dangerously close to 1/5 resonance!"];
```

will send the `alarm` client the warning message whenever the value of `get-tune.ready` lies between 9.18 and 9.21.

All relational operators have precedence just lower than `[]`'s (so they have precedence just higher than the various flavors of `&`).

The Glish "Control Panel" Tool

A particularly useful piece of the *Glish* system is the "control panel" tool, called `Con`, which provides a generalized user-interface for the system.

`Con` is a *Glish* client which is configured by sending it a `go` event whose value is the name of a configuration file. The file contains a series of lines describing the user-interface elements `Con` should provide. The simplest element is to display the most recent value of an event. For example, the line:

```
display "Horizontal and Vertical Tunes" tunes
```

configures `Con` to put up a display region labeled "Horizontal and Vertical Tunes" and to put in it the value of the last `tunes` event received by `Con`. The display will be updated every time `Con` receives a new `tunes` event. To add this to the `archive-tune` example above, we'd need to add `Con` to the `requires` line and then configure it with the name of a file containing the `display` line (and perhaps many other interface specifiers), using something like:

```
archive-tune.activate -> Con ["tunes.config"];
```

Previously we assumed that we had a `console` tool which every time it received a `display` would display the corresponding message:

```
get-tune --display-> console ["Tune values are:" get-tune];
```

This now changes to:

```
get-tune --tunes-> Con;
```

Similarly, `Con` could be used to display any or all of the shot number, timestamp, and most recent tunes.

Another user-interface element is a button. In the configuration file, the line

```
button "Reset CAMAC" reset
```

instructs `Con` to display a button labeled "Reset CAMAC", and whenever it is clicked on using the mouse `Con` will generate a `reset` event. This could then be passed on to `get-tune` (and, in turn, `read-bpms`) with an event expression like:

```
Con.reset --reset-> get-tune;
```

Button events can also have values associated with them. For example,

```
button "BPM Calibration" set-run-type bpmcal
button "Centroid kick" set-run-type kick
```

would make two buttons, both of which when pressed would generate `set-run-type` events but with different values. One could then have event expressions such as:

```
# if we're starting a new run, the tape archiver needs to
# reinitialize
Con.set-run-type --init-> tape_archive;

# and if it's a BPM calibration run, enable the BPM bump
# kickers
Con.set-run-type == "bpmcal" --enable->
    kicker-control ["bpm-bump"];
```

Another type of user-interface element which can be specified in the configuration file is an input window, in which whenever the user types a some text (followed by a carriage-return or a mouse-click), the text is sent as the value of the given event. For example,

```
input "Shot annotation" annotation
```

will put up a text-entry window with the prompt "Shot annotation". If the user types "looks like junk" in the window, `Con` will generate an `annotation` event whose value is "looks like junk". This could then in turn cause the annotation to be logged by the tape-archiver, for example.

The other style of user-interface currently provided is numeric input. A line in the configuration file like

```
numeric "Kicker Voltage (kV)" kicker-voltage 5 15
```

specifies that there should be an entry window with the given label. Whenever a number in the given range is typed into the window, `Con` will generate a `kicker-voltage` event with the corresponding value. The upper and lower bounds are optional and default to unbounded.

Finally, `Con` provides hooks for creating "raw" events; that is, one can put in the configuration file a line like:

```
rawbutton "Fake A Shot" read-bpms.camac-sun.got-shot "Region5"
```

which when pushed will generate a `got-shot` event with value `Region5` identical to what would be generated by the `read-bpms` client running on `camac-sun`. There is also a mechanism by which the user can type in the complete specification of a raw event (client, host, and event name) and have `Con` synthesize the event.

Often elements of the user-interface are conditional. For example, suppose we have a button which when pressed will archive the current shot. In this case archiving doesn't make sense unless we do, indeed, have a shot to archive. This dependency would be specified in the configuration file using a line like:

```
when got-shot    button "Archive shot" archive-shot
```

which will keep the button disabled until `Con` receives a `got-shot` event. There is a matching `unless` qualifier which disables the button upon receipt of the given event. Event values can also be tested. For example,

```
when kick-status = "enabled"  button "Kick the Beam" kick-it
```

will enable the “Kick the Beam” button only when the value of the last `kick-status` event received was “enabled”.

Currently only buttons can be made conditional but clearly one would like to have the capability for all types of user-interface elements.

Fig. 1 shows a `Con` display such as the ones used in the E778 experiment reported below, along with the configuration file used to generate the display.

Other Standard Glish Clients

In addition to `counter`, `std_client`, and `Con`, there are a number of other *Glish* utility clients:

- `hold_point` writes out a message specified by a `go` event and waits for the user to type a response. It then sends the response as the value of a `ready` event and awaits another `go` event.
- `cleanse` removes all pending events from the sequencer's queue.
- `send` is a simple interface for typing in raw messages to the sequencer.
- `timer` generates `ready` events every n seconds, where n is specified by a `go` event and can be fractional. It can be reconfigured by sending it another `go` event.
- `window` pops up a new window on the workstation screen and runs the given Unix process in it.

Support for Writing Glish Clients

Glish clients are written using what is currently a small C++[5] class library (clients must therefore be written in C++, but they can be simple “wrappers” which then call FORTRAN or C code). The base class in the library is “Client”. When a “Client” object is constructed the program waits for a `go` event. Once received, the object returns an argument list which corresponds to the values in the event. Client support the following operations: wait for the another event; send out an event with a given name and list of values; reconfigure when another `go` event is received. Consider a simple version of the `timer` client which is configured with the number of seconds for which it should sleep. It then waits for a `sleep` event and if it receives one it sleeps for the given amount of time and then generates a `ready` event with the value "timer expired". Then it awaits another `sleep` event. Here's the full code:

```
// simple timer client -
//     configured on "go" with the number of seconds
//     to sleep; goes ahead an sleeps when a "sleep"
//     is received; generates a "ready" upon waking up

#include "glish.h"
#include "Client.h"

main( int argc, char** argv )
    {
    Client c( argc, argv );
    // we have now received our "go" event and the values
    // have been put in the argument vector

    // convert first argument to number of seconds
    int num_seconds = atoi( argv[1] );

    ClientEvent* ce = c.NextEvent();
    while ( streq( ce->event_name, "sleep" ) )
        {
        sleep( num_seconds );
        c.PostEvent( "ready", "timer expired" );
        ce = c.NextEvent();
        }
    }
```

This client is particularly easy to write since it receives events synchronously. That is, it decides when it wants to wait for the next event and is willing to be blocked at that point until the event arrives. Often waiting for events in this fashion is too restrictive. There are two subclasses of Client for receiving events asynchronously. The first, PingClient, lets the client provide a routine which is called asynchronously whenever the client has a new message waiting to be read. The second, FlagPingClient, sets a flag specified by the client whenever the client has pending messages. The client can then periodically check the flag.

Implementation and Performance Issues

We implemented the *Glish* sequencer as a distributed Unix process. The code is roughly 2000 non-blank lines of C++ code and headers. The parts of the sequencer running on different hosts communicate with one another using remote procedure calls, and with their local clients using System V message queues. Because of this implementation *Glish* is presently not suited for tasks requiring events to be delivered between clients in real time ($< .1$ sec). Our assumption is that the *Glish* system will be layered on top of a rapid real-time system to provide less flexible but faster event delivery when required. It may also prove possible to implement the *Glish* system using different, higher-performance IPC and network communication protocols. *Glish* would then be suitable for lower-level control tasks too. We have not as yet explored this possibility.

The utility clients are all written in C++ and are quite small, from about 20 to 200 non-blank lines. The C++ class library for *Glish* clients is another 225 lines. Finally, `Con`, the control panel client, is about 500 lines of C++, built on top of a 1700 line graphical-interface class library. In general we have found C++ to provide an excellent methodology for building extensible libraries and applications. It proved extremely flexible: the entire prototype *Glish* system took a couple of weeks to put together.

Experience Using the Glish System

The *Glish* system was used for data acquisition, analysis, and control in a continuation of Tevatron Experiment E778[1]. The purpose of the experiment is to measure aspects of non-linear beam dynamics in the Fermilab Tevatron (in particular, searching for resonance islands). A typical data acquisition run involved:

- getting the accelerator in a desired configuration (energy, tunes, sextupole strengths, beam intensity);

- increasing the amplitude of beam oscillations by stepping through a range of kicker voltages;
- for each bump, digitizing anywhere from 10,000 to 1,000,000 turns worth of turn-by-turn data read by two pairs of horizontal and vertical BPM's[3];
- analyzing the data to see if the shot was good;
- if so, saving the shot information for eventual archival to an optical disk.

The entire data acquisition and on-the-spot data analysis for the experiment was done using the *Glish* system, for which the largest sequence used coordinated 43 clients and used about 120 event expressions. In addition the system was used to compute tunes, calibrate the BPM measurements (by measuring the BPM response to beam kicks of a known size), and measure chromaticity by changing the RF frequency and measuring the corresponding change in the tunes.

The *Glish* system ran on a group of three Sun 3 computers, one which acquired the BPM data via a CAMAC crate, another which controlled the sequencing and data archiving, and a third which ran the data analysis and display tools. The *Glish* system was connected to the Tevatron control system via a serial line between one of the Suns and a control system Vax. The connection provided the *Glish* system with readings from the control system database such as kicker strengths, nominal tunes, and beam intensities, and provided a way for the Suns to control the accelerator for operations such as kicking the beam and changing the RF frequency.

The E778 acquisition and analysis software uses a self-describing data standard[4] for communicating data between programs. The data-description is medium-independent; the data can reside on disk, in process memory, in shared memory, or in a database. Thus the programs can easily communicate data by generating events whose values are simply names identifying SDS datasets. A final and crucial component in the software system is Kaspar, a general-purpose dataset plotting program. Kaspar can attach to any SDS dataset. It then searches the dataset for any plottable elements (such as floating-point or integer arrays) and for each element produces a plot window which can then be expanded, compressed, measured, and correlated with other plot windows. A typical use of Kaspar was to display FFT's of beam position data to pick out tune spikes and chromatic sidebands. Another was to view blocks of turn-by-turn data in order to measure beam decoherence following the firing of a kicker.

During the E778 experiment the *Glish* system performed well. Many sequences were written and modified in what proved to be a simple and rapid process. We found the `Con` tool especially valuable, as it enabled physicists wholly unfamiliar with the *Glish* system to undertake the routine operations of the experiment. The software difficulties encountered almost always resided in portions of programs separate from the *Glish* system and the inter-program coordination it provided. It was clear, however, that there are a number of features which if added to the *Glish* language would make it easier to write large sequences, including: local and global variables; error and exception handling; a mechanism to restrict an event's scope to a small portion of a sequence; a mechanism to expand an event's scope beyond a sequence; more operators for manipulating event values; ways to check-point, hold, and recover sequences; more generalized parameterization of sequences; a mechanism to detect and eliminate event "races"; a "compiler" for automatically generating clients given a list of the events they generate and consume; and a graphical interface for dynamically editing sequences.

These are all areas we hope to continue developing as we gather more experience using *Glish*.

Future Directions

As mentioned above, there remains a good deal of work to be done on the *Glish* system to make it easier to use and more powerful. We find our initial experiences encouraging, and are now in the process of evaluating how to use the system to control the SSC Magnet Test Laboratory so we can acquire further experience with the system. Another promising area where *Glish* may be highly valuable is coordinating the pieces of large simulation systems. In particular, *Glish* could be used to good effect for integrating the many pieces of the software system which simulates the physics of the SSC[2]. By building the simulation using *Glish*, one again comes closer to the goal of ultimately creating a high-level control system by simply replacing a few, modular components of the simulation system.

References

[1] A. Chao et. al. Experimental Investigation of Nonlinear Dynamics in the Fermilab Tevatron. *Physical Review Letters*, 61(24), 1988.

[2] V. Paxson et. al. A Unified Approach to Building Accelerator Simulation Software for the SSC. In *Proceedings of the 1989 IEEE Particle Accelerator Conference, Chicago, IL*, 1989.

[3] S. Peggs, C. Saltmarsh, and R. Talman. Million Revolution Accelerator Beam Instrument for Logging and Evaluation. *SSC Central Design Group Note, SSC-169*, 1988.

[4] C. Saltmarsh. SDS Usage Documentation. SSC Laboratory Report in Preparation, 1989.

[5] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 1986.