A Software System for Modeling and Controlling Accelerator Physics Parameters at the Advanced Light Source *

L. Schachinger and V. Paxson
Lawrence Berkeley Laboratory
Berkeley, CA 94720

Abstract

We describe a software system used at the Advanced Light Source for accelerator physics studies and accelerator control. The system consists of a number of Unix processes that can be connected together in modular ways. Processes communicate using messages with a common data format, but processes do not know where their messages come from or go to, making each process easily replaceable by others using different algorithms, measurement techniques, or models. Some of the controls and correction functions we have implemented using the system are closed-orbit correction, continuous tune display, and Fourier analysis of turn-by-turn beam position monitor (BPM) data.

Introduction

Over the lifetime of an accelerator, and particularly in the commissioning phase, programs which measure and correct machine parameters can change significantly. For instance, at the ALS our current orbit correction algorithm is the local bump method, but we plan to implement an algorithm based on Singular Value Decomposition in the near future. As diagnostics come on line and are better understood, preferred methods for measuring a particular parameter change. Currently we use a model to calculate tunes from magnet currents, but soon we will read the tunes from a spectrum analyser, or perform an FFT of turn-by-turn data from the BPM's. These circumstances cry out for a modular, flexible approach, so that new correction algorithms or measurement techniques can be substituted and compared easily.

Toolbox Philosophy

We have long advocated a "toolbox" approach to building accelerator simulation and control software[1]. This approach emphasizes building applications by plugging together modular, single-function programs. The goal is to avoid monolithic, buggy, hard-to-maintain applications, and instead to stress having the flexibility to rapidly piece together new applications as the need arises.

Crucial to this approach is the modularity of the individual programs that comprise the toolbox. Each of these programs must be wholly self-contained; if we are to be able to connect the programs together in unforeseen ways, the programs must not assume anything about what other programs they might communicate with.

We can achieve this degree of modularity by making the programs *event-oriented*. By event-oriented, we mean that we write programs in terms of events they receive, telling them what to do; and events they generate, publishing the results of whatever they did. Each event is a name (e.g., "compute tune") and a value (typed data associated with the event). Event values can be quite large, as programs may have to communicate a large amount of data (e.g., computed β_x and β_y values at every BPM and corrector).

Programs do not know anything about where the events they receive came from, nor where the events they generate go to. In this way, programs remain completely modular. By making the events generated by one program become the events received by another, we can forge a new application from the two programs even though neither was written with any knowledge of the other.

The Glish Software Bus

An environment for connecting together these sorts of modular programs is sometimes called a *software bus*, in analogy with hardware busses that enable independently-designed hardware components to communicate. The software bus we use, called *Glish*[4], was designed with accelerator applications in mind.

While programs are written for use with Glish in an event-oriented style, Glish does not limit the names of the events used by a program nor the structure of the associated data. At first this might seem like granting the program writers too much freedom, since how can the programs communicate if they don't agree on data formats and naming conventions?

The answer lies in Glish's chief strength: Glish provides an interpreted scripting language, similar to that used in

*Work supported by Director, Office of Energy Research, Office of Basic Energy Sciences, Materials Sciences Division, U.S. Department of Energy under Contract Number DE-AC03-76SF00098.

0-7803-1203-1/93$03.00 © 1993 IEEE

shell programming, for piecing together applications from individual Glish programs. These scripts not only specify which programs to run, but can dynamically control what should be done whenever any of the programs generates a particular event. Here, "what should be done" includes possibly routing the event to another program (perhaps renaming it), and modifying the event's associated data. Thus Glish offers a powerful sort of "glue" that we can use both to connect disparate programs, and to overcome their incompatibilities. If, for example, we want to use programs written with different physical units, or sign conventions, or data structures, a Glish script can readily provide on-the-fly conversion between the two programs, without requiring any modification of any source code.

A final benefit of Glish is that it supports transparent networking. Glish programs can run on different hosts and never know that their events travel over a network.

Orbit Correction Application

One of our principle simulation and control applications for the ALS is *orbit correction*. The present application we describe here evolved from that described in [3].

On the face of it, correcting the orbit of an accelerator is a simple task. Given the machine's current trajectory, calculate the corrector settings necessary to flatten the trajectory; apply the new settings; and we're done. So simple that we might be tempted to write a single program to deal with the entire task.

In reality, though, many other factors enter into the application, and greatly complicate it. Correctors or monitors may be broken, disabled, or untrustworthy. We might need to use different hardware to read the first turn trajectory, before beam is stored, than the closed orbit, and a different correction algorithm in the two cases. We might want to average the position readings over a variable number of turns. Beam position monitors (BPMs) have offsets due to engineering errors, correctors have calibration factors for converting between radians of angle and amperes of current. We may have to apply corrections in steps, to avoid risking beam loss from overzealous correction. We may be able to use nominal phase and beta information for the accelerator, or want to calculate more precise values. Our "goal orbit" may change from a flat trajectory for stored beam, to a betatron oscillation when injecting.

Finally, we want a *single* application that can correct the orbit for both the ALS booster and the storage ring, taking into account all of the above factors. And we want to use this same application on-line, correcting the actual orbit, off-line for simulation using a modeling program instead of the actual hardware, and with data we previously archived, to try alternative correction strategies.

Figure 1 shows how we built the orbit correction application using Glish. The boxes along the left and right edges represent different Glish programs, all of which connect to the central software bus. The dotted box at the bottom represents static information that the Glish interpreter reads from data files and disseminates to those programs needing it.

The "Simulation / Accelerator" box represents one of two programs: either our modeling program (Teapot), for simulating orbit correction, or access to the actual accelerator hardware, for on-line orbit correction. The Glish script picks which of these two programs to use based on the script's run-time arguments.

Arrows indicate the events received and produced by each program. Note that there is *not* necessarily a one-to-one correspondence between an event produced by one program and an event sent to another program. Sometimes the Glish script itself deals directly with these events. For example, while the User Interface might request the current trajectory using a "`get orbit`" event, the Glish script decides whether to pass that event along to Simulation / Accelerator as a "`get closed orbit`" event, or whether to use the separate program for First Turn BPM Readouts. In the latter case, the script must send several different events to the program, one first to trigger the hardware, and then ones to read the X, Y, and signal sum values. Each of these elicits a separate event in response.

Picking between these two sources for the current orbit illustrates a key point: the system can accommodate two very different ways of getting the beam position data, and it does so transparently to all of the other programs involved in the application.

The orbit-correction application achieves all of the goals outlined above: we use it for both the ALS booster and the storage ring, both on-line, off-line to examine archived data, and off-line for simulation. A considerable amount of the application is done directly in the Glish script: converting between the units and sign conventions used by the different programs, incorporating BPM offsets and corrector calibration factors, averaging trajectories over multiple turns, applying fractional corrections, modifying first-turn readings based on the signal sum values, computing trajectory and correction statistics, and enforcing "fixed status" (e.g., "always off") for devices whose status is erroneously reported by the hardware.

Turnplot Application

Another application we built using Glish is *turnplot*, a program for analyzing beam position data[2]. Turnplot is structurally simpler than orbit correction, but maintains the property that it can be run on either the ALS booster or the storage ring, using either live hardware readings, tracking data produced by a modeling program, or previously archived data. Turnplot can analyze either a single, full turn of data (taken at each BPM), or an orbit scanned over every nth turn at a specified BPM.

Turnplot can display tracking data as a turn-by-turn point plot, as a phase space plot, or as X/Y data. In addition, we can FFT tracking data to identify probable X, Y,

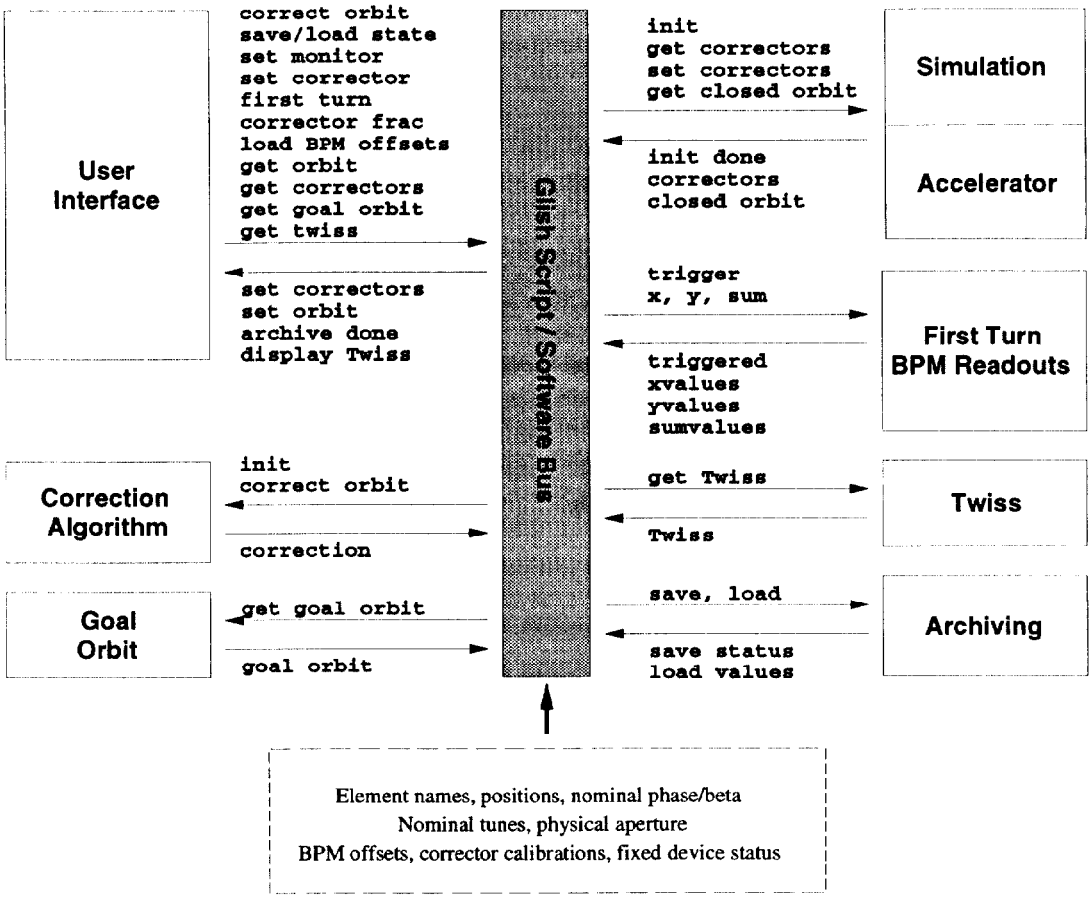

Figure 1: Structure of Orbit Correction Application

and synchrotron tunes, display harmonics associated with those tunes, select alternate FFT peaks if a peak found automatically appears unlikely, and display resonance plots for the identified tunes.

Tuneplot Application

A third application is *tuneplot*, for analyzing and controlling the machine tune. Tuneplot displays the current tune values on a resonance diagram. Like turnplot, tuneplot has no knowledge of where the tune values come from, so they can be changed transparently, including using a spectrum analyzer, the BPM hardware, and values computed from the present magnet currents. Because whenever tuneplot is sent a "`tune read back`" event it updates its display, we can use tuneplot to continuously display the tune, computed in "real time", without the program having any special provision for such a display.

Future Work

Other applications planned and in progress are one to measure and correct chromaticity, another to measure and correct betas (both by varying quadrupole strengths and measuring tunes, and by reading turn-by-turn data from all the BPM's while exciting a betatron oscillation), one to measure and correct dispersion, and one to measure and correct linear coupling.

References

[1] V. Paxson, C. Aragon, S. Peggs, C. Saltmarsh, and L. Schachinger, "A Unified Approach to Building Accelerator Simulation Software for the SSC," Proc. 1989 IEEE Particle Accelerator Conf., Chicago, IL.

[2] V. Paxson and L. Schachinger, "Turnplot: A Graphical Tool for Analyzing Tracking Data," Proc. 1991 IEEE Particle Accelerator Conf., San Francisco, CA.

[3] J. Bengtsson, E. Forest, H. Nishimura, and L. Schachinger, "Modeling in Control of the Advanced Light Source," Proc. 1991 IEEE Particle Accelerator Conf., San Francisco, CA.

[4] V. Paxson and C. Saltmarsh, "*Glish*: A User-Level Software Bus for Loosely-Coupled Distributed Systems," Proc. 1993 Winter USENIX Conf., San Diego, CA.