A Unified Approach to Building Accelerator Simulation Software for the SSC

Vern Paxson*, Cecilia Aragon, Steve Peggs,
Chris Saltmarsh, and Lindsay Schachinger
SSC Central Design Group**
c/o Lawrence Berkeley Laboratory
Berkeley, CA 94720

Abstract

To adequately simulate the physics and control of a complex accelerator requires a substantial number of programs which must present a uniform interface to both the user and the internal representation of the accelerator. If these programs are to be truly modular, so that their use can be orchestrated as needed, the specification of both their graphical and data interfaces must be carefully designed. We describe the state of such SSC simulation software, with emphasis on addressing these uniform interface needs by using a standardized data set format and object-oriented approaches to graphics and modeling.

Introduction

For the SSC to work requires thorough simulation of different designs and operational procedures[1]. Without detailed understanding of the machine's physics and how to deal with its various sources of error, it will prove impossible to effectively operate the accelerator. We are therefore developing a large software system to simulate the machine.

The goals of the simulation are three-fold: (1) to study the physics of different designs of the accelerator, (2) to develop high-level operational experience of controlling the machine in the face of anticipated errors, and (3) to build the simulation in such a way that it can later be used with the real accelerator, transcending pure simulation. For the SSC, the programs embodying the physics for the simulation span a large range of different types of models of the accelerator, different input and output formats, and different hardware. In addition, the simulations include highly graphical interfaces to enable the physicist using them to visualize aspects of the machine's behavior and thereby build intuition. The more effective these interfaces, the more effective the simulation.

When trying to develop a large, unified body of simulation software such as this, a number of software engineering issues arise:

- How do the different parts of the simulation talk to one another? Different modeling programs written by different people will have different input and output formats. Large quantities of data must be somehow coherently managed and accessible across heterogeneous networks. How does one deal with this without being overwhelmed with unwieldy ASCII files, cryptic command sequences, and error-prone drudgery?
- How to avoid spending all one's time trying to write effective interactive graphics for the simulation? How to sustain a uniform user-interface across the different simulations? The benefits of being able to visualize and directly manipulate the simulation's workings are enormous, but the software investment can be very large. One can't simply build a general high-level graphics library and expect to then quickly piece together whole interfaces from it. Effective user-interfaces tend to be *similar* to one another but not *identical*, making pre-canned solutions such as libraries inadequate. Often one winds up copying code from existing interfaces and then modifying it somewhat to suit the task at hand, leading to the horrific problems of maintaining a mass of almost-but-not-quite-duplicated code.
- How to design the simulation so that in the future it can operate on different models of the accelerator? What hope can one have of truly "plugging in" to the real machine, and turning the simulation into high-level control? Unless one begins early with these goals in mind, the simulation programs run a great danger of having wired into their innards all the global variables of the one modeling program currently at hand.

The body of this paper expands on the following approaches to addressing these issues:

To deal with the problem of interconnecting disparate programs and transparently moving distributed data across heterogeneous networks, we have developed a standard data format in which data objects are "self-describing," i.e., contain information about their structure as well as the actual data. Coupled with a distributed database[2], this will provide the backbone of a *software bus*–a common, machine-independent protocol which different programs can be plugged into to talk to one another.

The problem of being able to rapidly create new user-interfaces by specifying differences between them and existing ones, and to sustain a uniform user-interface across the entire host of simulation software, is especially amenable to object-oriented approaches. Rather than building a graphics subroutine library, we are developing a hierarchy of graphics *classes*, which can be readily extended and modified without duplicating code.

Finally, the general problem of developing model-independent simulations is also very well-suited to an object-oriented approach. By using classes to abstract the components of the accelerator simulation, we can build a system which will, without change, work with today's modeling program, tomorrow's modeling program, and, eventually, the real accelerator.

Taken together, these building blocks can provide the basis for a highly effective and flexible simulation system.

Standardized Data Sets

An integral part of our software system architecture is a uniform way to represent and communicate data. The solution must address several needs:

- data must be *self-describing*. That is, it must contain an internal description of its format, both low-level ("array

* Lawrence Berkeley Laboratory. Work supported in part by the United States Department of Energy under Contract Number DE-AC03-76SF00098.
** Operated by the Universities Research Association, Inc. for the U. S. Department of Energy.

of 512 doubles") and high-level ("aggregate named 'Twiss parameters', consisting of ...");

- the format must impose minimal overhead. Reading large data sets should be as fast or nearly as fast as directly reading binary data. The data must retain maximum precision;
- data must be readily transportable across heterogeneous networks, and in a transparent fashion (no explicit data conversions or network manipulations necessary);
- the data format must not be specific to disk files, but allow for alternate representations such as database entities, shared memory, and distributed access.

To this end we have developed an initial design and implementation of a *Self-describing Data Standard* (SDS[3]), which meets the above needs. The SDS library is callable from C, FORTRAN, and C++ programs, and currently supports Vax, 68000, and SPARC binary data formats. The first part of an SDS describes the byte-ordering, data types, and records in the data set. The remainder holds binary data, which is not converted to a "generic" format but remains in its native representation along with enough information to convert it to other representations.

To date SDS has been used: to take turn-by-turn data on the Tevatron for analysis on Sun workstations; to take magnet quench data on Vax computers running VMS, also for analysis on Suns (running Unix); and to represent the SSC lattice optics and multipole errors for communication between the thin-element Teapot[4] modeling program and the differential-algebra XMAP[5] modeling program. The data sets can reside on disk or tape, in the distributed database, in shared memory, or in process memory. Tools exist to list the contents of data sets, move them from one representation (e.g., disk) to another (e.g., shared memory), perform data transformations (e.g., FFT), provide graphical representations (various forms of plots), and to automatically generate data sets from a list of FORTRAN common blocks.

Our use of SDS will grow dramatically in the near future. We plan for it to become the medium of choice for all our data communication, thus providing a uniform way for disparate programs to communicate with one another. More SDS-related tools will be developed, such as a browser for exploring the high-level structure of a data set, filters for selecting parts of data sets and/or combining data sets, and additional transformation and analysis tools. Further work is also needed for making SDS-access fully transparent with respect to representation and networking.

An Object-Oriented Approach to Graphics

Having developed three generations of graphical interfaces for accelerator control[6][7][8] , we have come to appreciate how much effort one can spend writing and maintaining effective interfaces and how difficult it can be to directly reuse or build on large portions of existing ones. The last of these generations endeavoured to facilitate reuse by building a hierarchy of graphical "packages," each level of which supported the operations of lower levels plus additional functionality. For example, the `window` package implemented a simple window with an integer coordinate system and some line and text drawing functions. These windows had concepts of alignment with other windows, font sizes, colors, and mouse-clicks. The next level in the hierarchy, `world window`, was a "window" plus the concept of a world (floating-point) coordinate system. Similarly, `split world window` extended these to have a wrap-around point in the coordinate system, useful for representing objects such as accelerator rings where the beginning and end are at the same point. With this hierarchical approach, simple concepts such as "window" could branch out into different types of refinement, each of which could support the same basic functions (like "draw a line"). When writing new interfaces, one would select the necessary set of packages and then write the code to interconnect them.

We learned that trying to build such a hierarchy without support for it directly in the language (we were using C) is very difficult. One either winds up with a tangle of similar-sounding-but-different routine names (`window-draw-line()`; `world-window-draw-line()`; `split-world-window-draw-line()` ...), or packages which are second-class citizens–they don't support all the functionality of the package on which they're built, making interconnecting different packages very painful.

These problems cry out for object-oriented solutions. In an object-oriented language, the analog for a package is a *class*. Classes encapsulate both data associated with a concept (such as the coordinate system of a window) and functions (such as "draw a line"). An instance of a class is an *object*; one operates on objects by sending them messages telling them which function to perform on themselves, altering their internal state. Given some concept represented by a class, one can refine the concept by *deriving* a subclass from the original. Derived classes inherit all of their parent's functionality and state, plus they can introduce additional functionality. These extended classes are full-fledged citizens; any operation which can be performed on the base class can be performed on the extension. Furthermore, they can change how functions defined in the parent work for themselves. For example, a `world window` class can specify that *its* "draw a line" function means "same as for `window` except use the floating point coordinate system", overriding the previous definition. Then any routine written to deal with `window`'s can be handed `world window`'s as well. When the routine tells the object to draw a line, the correct version of the function is automatically used. Thus we gain two enormously useful advantages:

- New concepts can be created simply by specifying the *differences* between them and an existing concept; and,
- Routines can be written which will *automatically* work with future, unforeseen extensions to current classes, without requiring modification.

Passing messages around and automatically figuring out the right routines to call sounds potentially very inefficient. Fortunately there are object-oriented languages which are designed to maximize efficiency. One of these, C++[9] , is upwardly compatible with C and delivers the same high performance. Better still for our purposes, a graphical toolkit written in C++ is available for use under X Windows[10] (and possibly other platforms in the future), which gives immediate portability advantages[11] . Called InterViews[12] , the toolkit provides roughly 75 classes for writing user interfaces. The classes are all highly extensible (as one would hope!). We are now redesigning the graphical interfaces to all our accelerator simulation software to use InterViews. By using one common toolkit we can ensure uniform

user interfaces throughout the entire body of graphics software. To date classes have been written for interactive data plots, including zooming, panning, selection of points of interest, different styles of plotting, and fitting curves to data points; and for simple ways to create buttons, menus, cursors, dialog boxes, and text messages (all derived from more general InterViews classes). Using these classes we have developed interfaces for interactive chromaticity plotting and correction; decoupling; beta, eta, and closed-orbit plots; and viewing turn-by-turn plots, phase space plots, and smear plots of tracking data. (See elsewhere in these proceedings[1] for examples.) Already our collection of classes enables us to rapidly construct interfaces. As the class library grows we anticipate being able to create more and more elaborate interfaces just as easily.

An Object-Oriented Approach to Simulation

One of the boons of using object-oriented graphics is that in the process it becomes apparent how well-suited the approach is to other software problems. In particular, the ability to define a concept as a class and then refine the concept in different ways meshes extremely well with the goal of creating flexible simulation software. We are presently developing a set of "Machine" classes which abstract the models used to simulate the accelerator. So far, classes representing tracking data have been developed, and the beginning of a class encapsulating the general functionality of the modeling programs (such as "compute tune", "get/set multipole strength", etc.) is underway. The former have been used to develop programs to compute smear and graphical interfaces for exploring turn-by-turn data; the latter now replace explicit calls to modeling program routines in simulation programs.

Once all simulations are written in terms of the modeling program class, we will be able to transparently "plug" different modeling programs into the entire simulation, and, ultimately, the database and control system of the actual accelerator. With this approach, we can develop simulation software for immediate use which will also be directly applicable to the subsequent high-level control of the real machine.

Summary

We are now developing a large body of software to simulate the physics and high-level operation of the SSC. If this software is to form a unified and flexible whole, we must solve a number of software engineering problems that will otherwise render the system so bulky as to become effectively useless. To this end, we envision (1) the Self-describing Data Standard as providing a "software bus" on which programs can be plugged in to talk with one another and data transparently moved across heterogeneous networks; (2) a library of InterViews-based graphics classes to provide a way to rapidly create new interfaces and to ensure uniformity across all our interfaces; and (3) a library of model/machine classes to enable us to generalize our simulation to a variety of models of the accelerator, and, ultimately, to the actual machine.

We have a vision of how a truly effective accelerator simulation and high-level control system might look. In it, the user orchestrates a suite of interactive simulation programs, calling forth those relevant to the task currently at hand. With each such view, the user can select an object, be it low-level such as an individual magnet or high-level such as a non-linear chromaticity curve, and then summon a list of relevant operations, or query the object regarding its status, its history, and its meaning, following cross-ties to related objects. The user can attach objects to "clip-boards" for later reference, or move them between views to see them from different simulation perspectives (for example, select the working point achieved by a modeling program using a linear model of the machine, send it to a different program which includes nonlinearities to see if it can also achieve it; take the results from both, drop them into the Twiss parameters view to see how the optics are effected; or drop them into the Tracking view and, once tracked, into the FFT view to see what the actual tunes are). The goal would be to liberate the user from the drudgery that makes mixing simulation programs tedious and error-prone, and to give the user different visual perspectives of the simulation, that they might synthesize the different models of the machine and build better intuition as to the overall picture.

On the surface this vision seems far-fetched, perhaps overwhelmingly expensive to implement. But given a software bus, an approach for developing extensible, uniform interfaces, and a way to abstract modeling programs to render simulation software independent of them, the cornerstones are all in place. On this foundation such a unified system can truly be built and turned into reality.

References

1. L. Schachinger, et. al., "Modeling the SSC," these proceedings.
2. E. Barr, S. Peggs, C. Saltmarsh, "Relational Databases for SSC Design and Control," these proceedings.
3. C. Saltmarsh, "SDS usage documentation," SSC report in preparation, Berkeley.
4. L. Schachinger and R. Talman, "TEAPOT. A Thin Element Accelerator Program for Optics and Tracking," Particle Accelerators **22**, 35 (1987).
5. J. Irwin and S. Peggs, "Application of Multivariable Maps to Lattice Design and Analysis," these proceedings.
6. V. Paxson, et. al., "A Scientific Workstation Operator Interface for Accelerator Control," 1987 IEEE PAC, p. 556, Washington, D.C.
7. L. Schachinger, "Interactive Global Decoupling of the SSC Injection Lattice," Proceedings of the European Particle Accelerator Conference, Rome, August 1988.
8. V. Paxson, S. Peggs, and L. Schachinger, "Interactive First Turn and Global Closed Orbit Correction in the SSC," Proceedings of the European Particle Accelerator Conference, Rome, August 1988.
9. B. Stroustrup, "The C++ Programming Language," Addison-Wesley, Reading, Massachusetts, 1986.
10. R. Scheifler, J. Gettys, "The X Window System," ACM Transactions on Graphics, No. 63, 1986.
11. V. Paxson and E. Theil, "Towards Portability in Model-Based Control Software," LBL-24723, November 1987.
12. M. Linton, et. al., "The Design and Implementation of InterViews," Proceedings of the USENIX C++ Workshop, Santa Fe, New Mexico, November 1987.