A UNIVERSAL POSTPROCESSING TOOLKIT FOR ACCELERATOR SIMULATION AND DATA ANALYSIS*

M. Borland, ANL/APS, Bldg 401, 9700 South Cass Avenue, Argonne, IL. USA

Abstract

The Self-Describing Data Sets (SDDS) toolkit comprises about 70 generally-applicable programs sharing a common data protocol. At the Advanced Photon Source (APS), SDDS performs the vast majority of operational data collection and processing, most data display functions, and many control functions. In addition, a number of accelerator simulation codes use SDDS for all postprocessing and data display. This has three principle advantages: First, simulation codes need not provide customized postprocessing tools, thus simplifying development and maintenance. Second, users can enhance code capabilities without changing the code itself, by adding SDDS-based pre- and postprocessing. Third, multiple codes can be used together more easily, by employing SDDS for data transfer and adaptation. Given its broad applicability, the SDDS file protocol is surprisingly simple, making it quite easy for simulations to generate SDDS-compliant data. This paper discusses the philosophy behind SDDS, contrasting it with some recent trends, and outlines the capabilities of the toolkit. The paper also gives examples of using SDDS for accelerator simulation.

1 INTRODUCTION

In recent years, we have seen a great increase in the number of accelerator simulation codes. Each code developer faces the same hurdles of developing a user interface and providing tools for the user to view and analyze code output. Many lack the time or expertise to create sophisticated interfaces and postprocessing, and so provide difficult interfaces and little in the way of postprocessing.

In the midst of this, there is no movement toward common standards that permit simulation codes to be used together and share postprocessors. The lack of such a development reinforces the rate of growth, because users often find it easier to write a new code than to work with one or more existing ones.

Simulation codes still rely on an inordinate amount of user interaction and manual work. The time spent by the user to prepare and analyze data from a simulation may exceed the running time by orders of magnitude. A root cause of this is the lack of generally-applicable, powerful postprocessing tools.

In this paper, I describe a system that solves these problems. I will start by discussing common problems with simulation codes. Following this, I describe the structure and advantages of the Self-Describing Data Sets (SDDS) system, which is based on modular tools, self-describing data, and scripts.

* Work supported by U.S. Department of Energy, Office of Basic Energy Sciences, under Contract No. W-31-109-ENG-38.

2 COMMON SIMULATION CODE PROBLEMS

Data Storage: One of the root problems with simulation codes is the wide variation in data storage methods. Much of the output from most codes is user-readable text or graphics. Such output is inherently difficult or impossible to subject to automatic processing, which forces the user to manually transcribe or translate data into another format. Ironically, by providing "user-friendly" text or graphics, the code developer can actually make the sophisticated user's job much harder.

Some code developers recognize these problems and provide data files intended for automatic processing. However, in many cases, these are for use with a dedicated postprocessor that provides simple, single-purpose graphics and perhaps simple analysis. The user may not be able to count on these files remaining the same between code revisions. Other code developers take the approach of making data files for a specific commerical data analysis or viewing product. Rarely does more than one code use the same approach, let alone use the same data protocol.

Finally, most data files are "fragile," in that format changes will break programs that use the protocol. For example, a code may output a table of numbers that is intended for automatic processing. A user or the developer may create another code that reads the output of the first. When, however, a new column of numbers is added to the output of the first code, the second code is no longer functional. This creates an unwillingness to upgrade, a multiplication of versions of the same code, and an inability to use codes cooperatively.

Postprocessing Support: The time and effort a user spends postprocessing data from a simulation is generally proportional to the number of simulation runs performed. This is largely due to the lack of analysis software that accepts the output of simulation codes directly. This is often true of codes that provide a GUI, since the user of such an interface is expected to guide program actions with the mouse.

The analysis provided by the typical simulation code is often limited and not customizable. The user cannot ask for new computations to be performed upon the code's existing output data. Graphical output is also generally very specific and predefined. If the user wants something different, he must try to transfer the simulation output to another program.

In a way, this is only sensible. For each developer of each simulation to provide sophisticated graphics and postprocessing for each simulation code would entail an enormous amount of extra work. Hence, it is likely that this state of

affairs will continue unless another approach is found.

User Programmability: Few codes allow the user to go beyond the "Prepare, Run, Read" or "Click, Run, Look" simulation cycle. For example, there is usually no way to automatically prepare a series of input data sets for a simulation code, followed by automatic collation of the results of the group of runs. This is available with the custom programming environments that some codes provide, but it is unreasonable to expect every code to have such an environment.

What is needed is the ability to treat each simulation code as a computational module that can be embedded within another process (e.g., another program or script). For this to work, the root problem of data storage chaos must first be addressed.

3 A SOLUTION

An ideal situation would be the following: 1. All simulation output should be ready-made for machine manipulation and computation. 2. A common set of powerful data processing and display tools should be used for all accelerator simulation codes. 3. The output of any code should be readily used as input to another. 4. It should be possible to enhance and automate the use of a code without changing the code itself.

A system that provides these advantages is available and has been in use at the APS since 1994. The Self-Describing Data Sets [1, 2] system is based on a relatively simple self-describing file protocol. A toolkit is available that comprises about 70 general-purpose programs that use this protocol. There is an additional toolkit of about 20 control-system programs [3, 4].

SDDS has enjoyed wide application at APS, beginning with commissioning of the APS injector and continuing through commissioning of the APS storage ring. It provides computational muscle for most of the high-level applications in use today on the APS controls system, including control, display, and analysis. It is used for almost all experimental data collection and analysis for machine studies. It is also used for much of the accelerator simulation done at APS. In this paper, I will address only the last of these applications.

4 SELF-DESCRIBING DATA AND THE SDDS PROTOCOL

The concept of "self-describing data" may be unfamiliar, but it is quite simple. Basically, self-describing data is identified and accessed *by name only*, in contrast with traditional data formats where data is accessed by position (e.g., column number). Positional access methods make for "fragile" data files by requiring one code to "know" the detailed organization of data from another code.

With self-describing files, the code uses the name of the data to request the data values using a subroutine call. The details of where the data is in the file are encoded in the self-describing file protocol and intepreted by the subroutines that manage such files. As long as the name of the data and the self-describing file protocol are not changed, this system is robust and the data will always be readable no matter what additional data is put in the file.

There are other advantages to using self-describing files. With self-describing files, programs can be generic and operate on *named* data; for example, a graphics program can plot data in general, rather than just beta functions or just phase space diagrams. Self-describing files may also include "meta-data," such as units, and data type. Using such meta-data makes for more robust programs. For example, codes can check units and data type of data before using it.

An SDDS file consists of three parts: a single line that identifies the file as an SDDS file; an ASCII header consisting of namelist-like entries that define the structure of the data; and zero or more instances of the structure defined by the header. The data itself may be in binary or ASCII format, as declared by the header. The structure defined by the header may contain scalar values, columns forming a table, and arbitrarily dimensioned arrays. The user can access the elements of the structure individually. There are no inherent limits on the number of elements, the length of the tables, or the number of dimensions or size of arrays. Any element may have one of six data types: single- or double-precision float-point, short or long integer, character, or character string.

A familiar example of an SDDS file would be a table of Twiss parameters with corresponding values for the tunes, chromaticities, etc. After four years of experience with SDDS, we've found that the vast majority of our data fits this simple model in a natural fashion.

5 PROGRAM TOOLKITS

A "toolkit" is a group of programs that can operate on the same type of entity. Such programs can perform the same type of action on numerous entities of the same type. Ideally, any program's output is usable as input to any other program, so that programs can be used in sequence to process data. This is the chief advantage of such a toolkit. Another advantage is decentralized expansion, since anyone can independently create a tool that operates on the same entities. Such additions *cannot* break existing tools or the applications that use them, in stark contrast to the common situation where a change to one program requires changes to other programs.

The SDDS toolkit [5] is a group of about 70 programs that operate on SDDS files. There are three types of SDDS programs: those that read and write SDDS files, called the "core toolkit"; those that read SDDS files and create non-SDDS data (e.g., text or graphics); and those that read non-SDDS data (e.g., CSV files) and create SDDS data.

Capabilities of the individual programs are many, including graphics, equation evaluation, statistics, histogramming, fitting, interpolation, integration, differentiation, signal processing, smoothing, peakfinding, matrix operations,

and creation of text printouts. Since most programs both read and write SDDS data, many operations can be used sequentially on the same data set. The user can thus create a processing chain that suits his particular needs using the SDDS toolkit programs as modules.

For example, one might want to simply plot some data (using the SDDS program `sddsplot`). However, one might want to FFT the data and plot to FFTs (`sddsfft`, `sddsplot`). Or one might want to FFT the data, smooth the FFTs, find the amplitudes of the peaks, histogram the amplitudes, then plot the histograms (`sddsfft`, `sddssmooth`, `sddspeakfind`, `sddshist`, `sddsplot`). The possible combinations are literally inexhaustible.

Note that creating human-readable output is only a small part of the SDDS toolkit. Graphics and text printouts, while important, should not drive the structure of a simulation code or determine the format of its output. Using SDDS, the simulation developer doesn't need to worry about what kind of graphics or printouts a user might want. SDDS allows the user to produce the graphics and printouts *he* wants.

To give some more explicit examples, suppose that one has an SDDS data file (`data.sdds`, say) containing turn-by-turn particle coordinates from a tracking simulation. Suppose that the data is in a table, with columns called `x`, `px`, `y`, and `py` giving the transverse phase space coordinates, a column called `Turn` giving the turn number, and a column called `delta` giving the fractional momentum offset. (Space does not permit showing the plots from these examples. Examples of plotting output, all directly from the SDDS toolkit and unaltered, are shown in the Figures in following sections.)

• Plot the horizontal phase space:

```
sddsplot -column=x,px -graph=dot \
 data.sdds
```

• Take the FFTs of the horizontal and vertical motion. Plot these together using variable line types on a log scale:

```
sddsfft -column=Turn,x,y \
 -window=hanning data.sdds data.fft \
sddsplot -column=f,FFT* \
 -graph=line,vary -mode=y=log data.fft
```

Note that the FFTs of `x` and `y` are automatically named `FFTx` and `FFTy`. Most of the core toolkit functions this way, providing predictable new names based on the names of the data being being processed. In some cases, the user must supply a new name or a template for new names.

• Suppose now that the file `data.sdds` contains multiple pages, with each page corresponding to a separate particle with a different initial momentum offset. The following sequence will give plots of x and y tune vs momentum:

```
sddsprocess data.sdds -pipe=out \
 -process=delta,first,delta0 \
 | sddsfft -column=Turn,x,y -pipe \
 | sddsprocess -pipe \
 -process=FFTx,max,xTune,posi,func=f \
 -process=FFTy,max,yTune,posi,func=f \
 | sddscollapse -pipe=in data.result \
sddsplot -column=delta0,?Tune \
 -separate data.result
```

While this example looks a little cryptic at first, it is easily understood and is much more terse than doing the same thing in a programming language. First, `sddsprocess` is used to create a parameter (`delta0`) containing the `first` value of the column `delta`. Second, the `x` and `y` values are FFT'd using `Turn` as the independent variable. Third, `sddsprocess` is used to find the `position` of the `maxima` in `FFTx` and `FFTy`, viewed as a function of `f` (the frequency from the FFT); the resulting parameters are called `xTune` and `yTune`. Fourth, `sddscollapse` is used to "throw away" the tabular data and collapse the parameter data across pages to create a new table, containing columns `delta0`, `xTune`, and `yTune`. Finally, the result is plotted.

In order to coordinate toolkit programs and create high-level applications, we often employ a script programming language. SDDS simplifies the development of data processing, while the script language provides an easy interface to the data processing algorithm. Using SDDS is much easier than using a programming language for data processing because the user need not worry about loops, variables, and all the other overhead of programming. SDDS hides all of this from the user and lets the user think in terms of the operations he needs to perform.

6 SIMULATION CODES AND SDDS

There are a number of codes that are completely or partially reliant on SDDS for data storage and processing. Among the "fully-compliant" codes are: `elegant` [6], a general-purpose accelerator simulation code; `shower` [7], an EGS4 wrapper for electron-gamma shower simulation; and `spiffe` [6], a fully-electromagnetic PIC code for rf gun simulation. These codes use SDDS for all output and most input. (For input, we use standard format for the accelerator lattice and each code has a custom command-based main input stream.)

As an illustration of the flexibility of SDDS, the code `elegant` uses SDDS for all data except the lattice file and command input. Data include: particle input and output for tracking; turn-by-turn particle data and statistics; FFTs of particle motion; beam moments, transport matrix, and Twiss parameters vs position; coordinates of lost particles; amplification factors; orbits, corrector strengths, and statistics; magnet strength output after tune or chromaticity correction; output of internally-generated error values; input of data for any element parameter; input of kicker waveforms; input of impedances and wake functions.

The three codes noted above have been used cooperatively, a task made much easier by the use of SDDS files. For example, `elegant` can be readily used to track the output of `spiffe` or `shower`. `shower` can be wedged between two `elegant` runs to create a simulation of beam transport and shower creation follwed by tracking of shower particles. While `elegant` knows the

names of the data in `spiffe` particle data output and can read it directly, `elegant` doesn't know anything about `shower` files (which don't use accelerator-type coordinates). Instead an SDDS-based script is used to produce `elegant`-convention 6-D particle data files from `shower`-convention files. This illustrates how the SDDS toolkit can be used to glue together two disparate programs that use SDDS files but differ in their mathematical convention for describing results. A number of real-world projects have made use of this technique. These include rf gun design and transport line optimization [8] and multilayer positron target design [9].

It isn't necessary for a code to be fully SDDS-compliant in order to reap some of the advantages of SDDS. Several existing accelerator codes (including ABCI [10] and TDA3D [11]) have been modified to create SDDS ASCII files [12]. This is quite easy and gives any code so modified instant access to sophisticated postprocessing and graphics.

For some codes (MAD [13] and MAFIA [14, 15]), we have special-purpose output conversion tools available to convert the code's output into SDDS. For other codes (ACCSIM [16], RACETRACK [17], and GINGER [18]) we can convert to SDDS by prepending an SDDS header to the output data [12].

7 EXAMPLES

7.1 Enhancing a Code Using SDDS

The code `elegant` performs 6-D canonical tracking of particles, but is unable to compute tune as a function of momentum. It is straightforward to add this capability without modifying `elegant`. The items in parentheses are the names of the SDDS toolkit programs used in each step. The reader may notice a similarity with an example provided above.

1. Prepare input particle coordinates having a range of $\Delta P/P$ values and small x and y starting values (`sddssequence`, `sddsprocess`).
2. Run `elegant` to track each particle in succession.
3. FFT the turn-by-turn x and y coordinates of each particle (`sddsfft`).
4. Associate each pair of FFTs with the $\Delta P/P$ value (`sddsxref`).
5. Find the position of the peaks of each FFT, i.e., the tunes (`sddsprocess`).
6. Plot the result (`sddsplot`). Figure 1 shows the plot as it is produced by `sddsplot`.

Based on this, another script was developed to perform chromaticity correction from the results of tracking. Such sequences can be placed into scripts and used easily on different lattices. These examples show clearly how one can enhance a simulation without modifying it or having any knowledge of the source code. This is true of any code that allows fully automated postprocessing of data and preparation of input. One of the great strengths of SDDS is that it provides this capability for any compliant code.

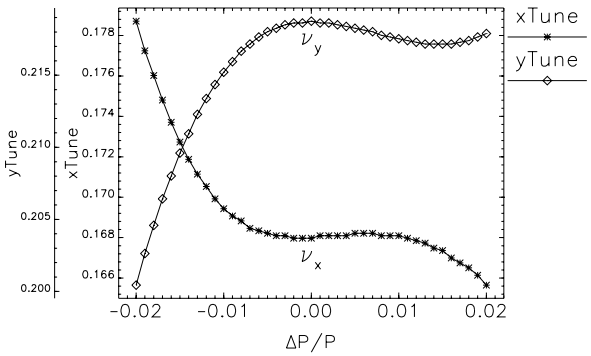

Figure 1: `elegant` tracking data postprocessed using SDDS

7.2 Scalable Simulation and Postprocessing

Often in the design stage of an accelerator, one runs simulations with random perturbations of magnet positions, strengths, and other parameters. Typically only 10 or 20 simulations are done, because it is very time-intensive to set up, run, and review the results of a single simulation. With SDDS, this impediment is removed, because all analysis can be done with an SDDS-based script. Once a script is written to postprocess the results of a single run, it is easily adapted to postprocess the results of an arbitrary number of runs.

For example, I simulated 1000 randomized configurations of the APS Positron Accumulator Ring [19] with alignment and strength errors, plus orbit, tune, and chromaticity correction. This was done by simulating 50 randomized configurations on each of 20 workstations. Each workstation performed 50 simulations and produced a set of SDDS files. These results were collated, processed, and displayed using a script employing the SDDS toolkit. Two examples of the results of this script are shown in Figures 2 and 3. (These plots are produced with `sddsplot` and are shown exactly as produced by that program.)

Using 1000 randomized configurations gives distributions of machine properties that simply aren't available with the small number of seeds used in most work. The use of SDDS to support quality simulations like this is not confined to any specific program, but is possible with any program that writes SDDS data (or for which an automatic conversion method exists from native data to SDDS data). In contrast to the typical situation, the amount of human effort involved with the SDDS-based approach is completely independent of the number of random seeds used. Further, the relatively small effort invested in the script pays off whenever simulation of another machine or lattice is needed.

7.3 Top-Up Safety Tracking [20]

Top-up operation, wherein beam is injected into a synchrotron light source storage ring with shutters open and beam available to users, is a high-priority goal at the APS. One issue with top-up operation is whether, due to a full or

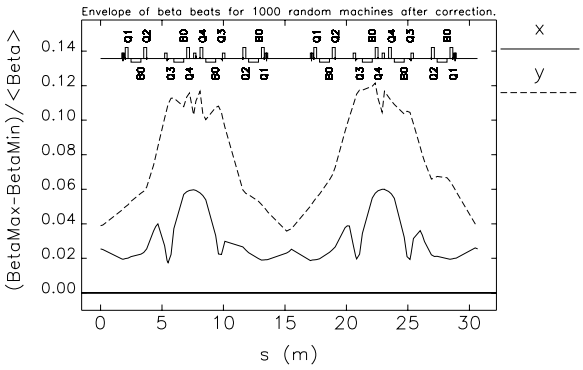

Figure 2: Beta functions from 1000 random machine simulations processed using SDDS

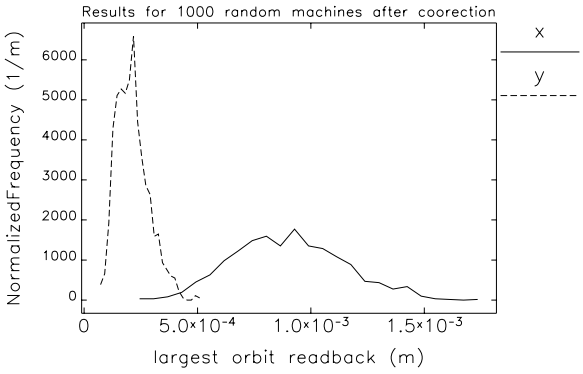

Figure 3: Orbit correction results from 1000 random machine simulations processed using SDDS

partial dipole short, injected beam could be extracted down a photon beamline while maintaining stored beam. If so, then the safety protocol planned by APS to protect users from direct injected beam would be inadequate.

Space does not permit going into the tracking studies in great detail. The approach was to simulate the existence of stored beam and the extraction of injected beam under the same conditions. The goal is to show that not only are the two incompatible, but that there is a gap (expressed in fractional strength error of a dipole) between the conditions that allow one and the conditions that allow the other. The simulations must include scenarios that cover multiple lattices, multiple tunes, and failures of other, nearby magnets in the most unfavorable way.

The location of apertures in each ring sector is part of the input for this endeavor. Each aperture configuration requires about 570 runs of `elegant` to evaluate the failure scenarios we devised. Frequently, one simulation is set up using the results of a prior one (e.g., an injected beam simulation uses the results of a stored-beam simulation that employed orbit correction to find a worst-case dipole kick from a supposed quadrupole short). A single script is used to submit the jobs, which are grouped in directories by scenario type. Each scenario directory has an individual script that prepares and submits jobs. Using 20 workstations, it takes two to three days to run a configuration.

A single script postprocesses the data. This script in fact runs a series of scenario-specific scripts. No manual collation or analysis is required. The results of the 570-odd simulations are reduced to a single number (the safety gap) plus an SDDS file containing summary data for each scenario. If a particular scenario needs to be investigated, the scenario-specific script can be used to make plots.

When faced with such a complex project, many would begin by developing a new simulation code. However, the SDDS-based approach allowed us to use an existing code and complete the simulations in a far more timely fashion.

8 REFERENCES

[1] M. Borland, "A Self-Describing File Protocol for Simulation Integration and Shared Postprocessors," Proc. 1995 PAC, May 1-5, 1995, Dallas, Texas, pp. 2184-2186 (1996).

[2] M. Borland, L. Emery, "The Self-Describing Data Sets File Protocol and Toolkit", Proc. 1995 ICALEPCS, Oct. 30-Nov. 3, 1995, Chicago, Illinois, pp. 653-662 (1997).

[3] L. Emery, M. Borland, "Commissioning Software Tools at the Advanced Photon Source," Proc. 1995 PAC, pp. 2238-2240 (1996).

[4] M. Borland, L. Emery, N. Sereno, "Doing Accelerator Physics Using SDDS, UNIX, and EPICS," Proc. 1995 ICALEPCS, pp. 382-391 (1997).

[5] M. Borland, "User's Guide for the SDDS Toolkit Version 1.8," http://www.aps.anl.gov/asd/oag/manuals/SDDStoolkit/SDDStoolkit.html

[6] M. Borland, ANL/APS, unpublished program.

[7] L. Emery, ANL/APS, unpublished program.

[8] J. Lewellen, ANL/APS, private communication.

[9] M. White, E. Lessner, "Slow Positron Target Concepts for the APS Linear Accelerator," Proc. 11th International Conf. of Positron Annihilation, Material Sciences Forum Volumes 255-257, pp. 778-780 (1997).

[10] Y.-H. Chin, CERN-SL-94-02-AP, February 1994.

[11] T.-M. Tran, J. S. Wuertele, "TDA -A Three-Dimensional Axisymmetric Code for Free-Electron Laser Simulation," Computer Physics Comm. 54, p. 263, 1989.

[12] Y.-C. Chae, ANL/APS, private communcation.

[13] F. C. Iselin, "Status of MAD (Version 8.5) and Future Plans," CERN-SL-92-46-AP, Sep 1992.

[14] M. Dehler *et. al.*, "Status and Future of the 3-D MAFIA Group of Codes," COMPUMAG Conf. on the Computation of Electromagnetic Fields, Tokyo, Japan, Sep 3-7, 1989.

[15] L. Emery, ANL/APS, private communication.

[16] F.W. Jones *et. al.*, "ACCSIM: A Program to Simulate the Accumulation of Intense Proton Beams," Int. Conf. on High Energy Accelerators, Tsukuba, Japan, Aug 22-26, 1989.

[17] F. Iazzourene *et. al.*, "RACETRACK USER'S GUIDE VERSION 4.01," Sincrotrone Trieste Report ST/M-97/7, July 1992.

[18] W.M. Fawley, "An Informal Manual for GINGER and its post-processor XPLOTGIN," LBID-2141, December 1995.

[19] M. Borland, "Commissioning of the Argonne Positron Accumulator Ring," Proc 1995 PAC, pp. 287-289 (1996).

[20] M. Borland, L. Emery, private communications.