A Self-Describing File Protocol for Simulation Integration and Shared Postprocessors*

M. Borland
Advanced Photon Source, Argonne National Laboratory
9700 South Cass Avenue, Argonne, Illinois 60439 USA

Abstract

A typical accelerator physics code uses a combination of text output, unformatted output, and special-purpose graphics to present results to the user. Most users must learn multiple graphics and postprocessing systems; many resort to manual extraction of data from text output, creation of customized postprocessing programs, and even modification of the simulation code. This situation slows research, results in duplication of effort, hampers unforeseen use of simulation output, and makes program upgrades potentially traumatic. This paper discusses the design and use of a self-describing file protocol that addresses these problems. An extensive toolkit of generic postprocessing programs, including sophisticated graphics, is available. This system has been used for most of the data collection for Advanced Photon Source (APS) commissioning, and is incorporated into a number of simulation codes.

I. INTRODUCTION

The structure of a typical accelerator physics code has changed little since the days when the only readily available output was the printout. Most codes still concentrate on this type of text-oriented output. The user wishing to postprocess a printout often ends up reading the data from paper or from a computer screen, and doing essentially manual computations, a procedure that is reminiscent of using tables of trigonometric functions and logarithms. A more sophisticated user may write a postprocessing program that reads the printout and performs computations. Unfortunately, the printout is almost necessarily either difficult to read or else difficult to parse. The printout typically mixes many types of data, making a robust postprocessor difficult to write. The user is frequently forced to resort to a text editor to extract the data of interest, which is time-consuming and error-prone. If the user succeeds in writing a postprocessor, he may find that it doesn't work with the next version of the simulation code, due to changes by the simulation's authors. This applies equally if the postprocessor is another simulation code, which explains why so few of the accelerator codes in existence today are able to use one another's output.

To make a bad situation worse, the user's effort to create a parser for a postprocessor must be duplicated for every code he uses, since there is no standardization of text or binary formats. If the user wishes to take advantage of the special features of two different simulation codes, he must write two different parsers. Many users find it easier to modify the simulation code itself rather than write a postprocessor. This again makes upgrades painful (since the modifications must be made to each new version), and results in proliferation of nonstandard versions of the simulation code.

In an effort to solve some of these problems, code writers often supply a special-purpose postprocessor that does what they anticipate the user will need to do. This postprocessor may do little more than graphics, or it may do mathematical operations on the data; it is unlikely to be as sophisticated in these functions as commercially-available or generic packages. It is also unlikely to be compatible with similar postprocessors for other codes in terms of data formats or commands. These multiple postprocessors frequently duplicate each other's functions (e.g., graphics), which wastes effort. Further, the user is not free to exploit the special features of a particular postprocessor with data from an unrelated simulation. Finally, if the user needs to go beyond what the supplied postprocessor allows, he encounters the problems discussed above.

This paper discusses use of the "Self Describing Data Sets" (SDDS) file protocol to eliminate these problems.

II. SELF-DESCRIBING DATA

The concept of self-describing data starts from the recognition that a scientist typically associates a number of attributes with data: 1. The name by which the data is known. 2. The units of the data, if any. 3. The meaning of the data (i.e., a description). 4. A mathematical symbol to represent the data, if appropriate. 5. The type of data (e.g., floating-point).

A true self-describing file protocol (SDFP) should incorporate these attributes. The user of self-describing data obtains the data only by name. The user need not know, for example, which column of a table a quantity appears in or how the data is formatted. This is the crucial feature of self-describing data, as it enables one to avoid the above-mentioned pitfalls.

For the purpose of discussion, imagine some data that can be organized into a single table. For example, the data could be position, Twiss parameters, element name, etc., along an accelerator. Any program accessing the data would do so by name, using routines supplied by the creator of the SDFP. If the program needed only certain data (e.g., position and horizontal beta function), it would request only that data. The presence of additional data (e.g., dispersion), would be irrelevant. Further, the program need not know the source of the data—it could be from any source, from direct user input into a file to output from a simulation code.

If all of the presently-available accelerator codes that perform comparable computations employed the same SDFP for their output, users and programs could access data without regard for which code it was from. Programs would not need to be custom-designed to provide output to each other in order to work together. This would allow users to combine programs in ways not planned by the programs' creators. The only con-

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

straint would be that the codes used the appropriate names for quantities. (In practice, this restriction can be reduced by designing programs to request data under several different but equivalent names.)

The example of Twiss parameter output is illustrative for another reason. Most codes that compute Twiss parameters print the results in a single large table with 132-column lines. This forces truncation to (typically) three to six significant figures, in order to fit all of the data for each element on a single row of the table. Some programs provide a second output file that contains the data to full precision. Using an SDFP, neither the lack of precision nor the duplication of data would be necessary. Note that the user rarely looks at all of the columns present in a typical printout; for the user who requires a printout, a tool to take an SDFP file and create a customized printout containing only the columns of interest would be more satisfactory. Such a tool could be completely generic, so that it could be used with any program compliant with the SDFP.

III. THE SDDS DATA MODEL

The principles elucidated in the previous section have been implemented in the SDDS protocol. Any SDFP implementation makes assumptions about what type of data will be stored and how it will be arranged. These assumptions comprise the data model for the protocol. At the highest level, the SDDS model organizes data into a series of "data pages." Each page of any file must contain the same elements, but may contain different specific data. For example, each page could contain the Twiss parameters for a different accelerator, or for a different tuning of the same accelerator.

Within each page, the following classes of data are recognized: 1. Tabular data, consisting of an arbitrary number of rows of mixed-type data. The data in the table is referred to by the name of the column. The same columns are expected for each page of the file. 2. Array data, where each element is of fixed but arbitrary dimension (i.e., an arbitrary number of array indices is allowed). Arrays are accessed separately, but may be placed in groups. The size of an array may vary from page to page. 3. Parameter data, consisting of single values that may either be fixed throughout the file, or vary from page to page. A parameter is essentially an array containing a single value, but has simplified access.

Any element of these data classes may have one of the following C-language data types: float, double, short, long, char, and char *. These are, respectively, single and double-precision floating point, short and long integers, single characters, and character strings. All of these types may be mixed in the tabular data section, but each column must contain data of fixed type. Note that SDDS has no restrictions on the numbers of each type of element, on the length of the tabular data, on the dimension of arrays, on the size of arrays, or on the number of characters in string data elements.

To continue the example of storing Twiss parameters, one might create an SDDS file containing the following: 1. Parameters: The name of the lattice, the tunes, the chromaticities, the acceptances, etc. 2. Columns: The element name, the position, the Twiss parameters, etc. 3. Arrays: The response matrix, matrices for tune and chromaticity adjustment, etc.

Other self-describing protocols are less restrictive than SDDS in that the data model is more flexible. While this is more powerful, it is considerably more complicated for both the program developer and user, which probably explains why existing SDFPs are not widely used. Experience with SDDS shows that there are very few instances where a more complicated model is required. At worst, the user may need to store disparate data in different, parallel files; this actually has the advantage of making the data easier to access. Some examples of data stored in SDDS at APS will be given in later sections.

Another advantage of SDDS over more complicated protocols is that the data may be either ASCII or unformatted (i.e., "binary"). The SDDS header (which describes the structure of the data pages), is in ASCII and has a familiar namelist format. It is a simple matter to create an SDDS file using print statements in a program, giving immediate access to a wide range of SDDS tools. The SDDS header has features to make it easy to convert existing text data into SDDS protocol; often, one merely creates a header and attaches it to the top of the file. These statements are not true of other SDFPs that I know of.

The SDDS header incorporates a protocol and version identification string, allowing determination that a given file is in SDDS protocol, and providing the version of the protocol. This permits upgrades of the protocol itself without disrupting users by making existing data or programs obsolete.

IV. THE SDDS TOOLKIT

A further break with traditional methods in accelerator simulation is the introduction of the toolkit concept for postprocessing [1]. A "toolkit" is a group of independent but cooperative programs. Traditionally, postprocessing has involved writing single- or few-purpose programs devoted to a single simulation code, in spite of the fact that the operations performed by many postprocessors are essentially identical. With the use of SDDS, it makes more sense to write generic programs that perform operations on data referred to by name.

While toolkit programs are shared by users, development is decentralized. The only requirement for an SDDS toolkit program is that it read and/or write SDDS files, so anyone may contribute programs. This is an advantage over "all-in-one" packages, which force the user to import data into a single program and work within a centrally-controlled environment. In contrast, toolkit programs are combined through use of the command shell and through command "scripts"; often, several programs are used sequentially on one or more data files. SDDS-compliant programs automatically work together by virtue of the common "language" of SDDS protocol, with little or no planning on the part of code developers.

The SDDS Toolkit is a growing group of about 35 programs that use SDDS files. Most accept SDDS input and produce SDDS output. In contrast to recent trends toward inefficient, tedious, and restrictive graphical user interfaces (GUIs), existing SDDS programs are accessed from the command line. While a GUI is sometimes helpful to the novice or occasional user, it is rarely of benefit to the serious user except when the program is inherently graphical. Further, command-line tools can be included in scripts that require no user interaction. This permits assembly of custom postprocessing commands for repetitious tasks. Much of the data processing for APS commissioning is handled by such scripts [2].

Space permits mentioning only a portion of the Toolkit: **sddsplot** is a flexible, commercial-quality, device-independent graphics program. **sddscontour** is a graphics program for making contour and density maps of data. (Both of these are command-line driven, but bring up a GUI under X Windows.) **sdds2spreadsheet** and **sddsprintout** convert SDDS output to spreadsheet input or customized printouts. **sddsprocess** is a powerful data processing program that, among other features, permits computation, scanning, editing, statistics, and selection operations. **sddsgfit** and **sddsexpfit** do Gaussian and exponential fits. **sddsfft** does Fast-Fourier Transforms. **sddshist** and **sddshist2d** do one- and two-dimensional histogramming. **sddscorrelate** analyzes data for correlations, while **sddsoutlier** eliminates statistical outliers. **sddssort** sorts data by multiple user-specified criteria. **sddsxref** transfers data between files, with optional cross-referencing. **sddschanges** and **sddsenvelope** analyze data over multiple data pages. **sddsconvert** converts between binary and ASCII modes, as well as renaming and deleting elements.

V. SDDS-COMPLIANT SIMULATION CODES

A number of simulation codes have been converted to write and/or read SDDS files. Three distinct related programs are highlighted, with mention of how SDDS integrates them.

elegant [3] is an accelerator simulation code using matrix methods (up to second order), canonical integration, and numerical integration, with MAD-format lattice input. **elegant** provides up to 25 different SDDS output files containing widely varying types of data. While producing many separate files may seem cumbersome, it greatly simplifies the use of the data and the internal organization of the simulation. A typical simulation would result in the production of several of these.

For example, one could simulate a transport line with an arbitrary number of random perturbation sets. For each perturbation set, **elegant** could output the Twiss parameters and the transfer matrix, beam sizes and centroids from tracking, particle coordinates at specified locations, information on particles lost on apertures, trajectory predictions before and after correction, and more. **elegant** also accepts SDDS data as input. For example, if one asks for logs of random perturbations used during a simulation, one can have **elegant** read the perturbations for use in another simulation; one can generate the perturbation input by other means, e.g., from magnetic measurement or survey data. **elegant** will accept its own particle coordinate output as input, allowing multistage tracking; the same output can be postprocessed with any SDDS tool, or examined graphically with **sddsplot**.

shower [4] is a C program that provides an easy-to-use interface to the **EGS4** (Electron-Gamma Shower) program. **shower** not only produces SDDS output of multi-specie shower products, but will also read such output to allow multi-stage simulation.

spiffe [5] is an electromagnetic field and particle-in-cell code used for rf gun simulation. Output includes snapshots of particle coordinates at constant time or position, electromagnetic fields at probe points, electromagnetic field maps, and the cavity boundary.

All three programs use only **sddsplot** and **sddscontour** for graphics. None requires any special-purpose postprocessing codes. **sddsprocess** can be used to translate the differing particle coordinate conventions of the three codes to permit tracking **shower** or **spiffe** output with **elegant**, or using **elegant** output as input to **shower**.

VI. SDDS-COMPLIANT EPICS APPLICATIONS

The control system used for the APS, known as the Experimental Physics and Industrial Control System (EPICS) [6], is used at a number of accelerator facilities in the United States. A number of “add-on” applications that use EPICS facilities have been developed, and are used for APS commissioning and operations. For example, **sddsexperiment** is a program that performs generic experiments on EPICS process variables (PVs); this includes changing PVs, reading back and analyzing PVs, and executing subprocesses. One use is measurement of response matrices. **sddsmonitor** and **sddsvmonitor** are EPICS monitoring programs, with sophisticated features like glitch- and event-triggered data logging.

With very few exceptions, SDDS is used for all accelerator commissioning data. For example, SDDS is used for: saving and restoring machine configurations; magnet conditioning instructions; GPIB device configurations; data from digital oscilloscopes and spectrum analyzers; machine history data; experimental data collected from process variables; data for generalized feedback on process variables [2]; and response matrices (from **elegant** or experiment) for orbit correction.

VII. OBTAINING CODE AND MANUALS

A distribution version of the SDDS library and Toolkit is expected to be available shortly after conference time. Manuals will be available as hypertext via World Wide Web, and in Postscript format. Details may be obtained by contacting the author at borland@aps.anl.gov.

VIII. REFERENCES

[1] M. Borland, “A High-Brightness Thermionic Microwave Gun,” Stanford Ph.D. Thesis, 1991, appendix A.

[2] L. Emery, “Commissioning Software Tools at the Advanced Photon Source,” these proceedings.

[3] M. Borland, “Users Manual for **elegant,**” APS LS-231, May 6, 1993.

[4] L. Emery, “Beam Simulation and Radiation Dose Calculation at the Advanced Photon Source with **shower**, an EGS4 Interface,” these proceedings.

[5] M. Borland, unpublished program.

[6] L. R. Dalesio, et. al., “EPICS Architecture,” ICALEPS 1991, pp. 278-281, 1991.