ACCELERATOR DESCRIPTION FORMATS*

Nikolay Malitsky, BNL, Upton, NY 11973, U.S.A.
Richard Talman, Cornell University, Ithaca, NY 14853, U.S.A.

Abstract

Being an integral part of accelerator software, accelerator description aims to provide an external representation of an accelerator's internal model and associative effects. As a result, the choice of description formats is driven by the scope of accelerator applications and is usually implemented as a tradeoff between various requirements: completeness and extensibility, user and developer orientation, and others. Moreover, an optimal solution does not remain static but instead evolves with new project tasks and computer technologies. This talk presents an overview of several approaches, the evolution of accelerator description formats, and a comparison with similar efforts in the neighboring high-energy physics domain. Following the Accelerator-Algorithm-Probe pattern, we will conclude with the next logical step, Accelerator Propagator Description Format (APDF), providing a flexible approach for specifying associations between physical elements and evolution algorithms most appropriate for the immediate tasks.

INTRODUCTION

Since accelerators are relatively complicated devices, their description is both important and complicated. Different descriptions are appropriate for different purposes. This article considers only descriptions that are appropriate for analyzing the behavior of beams of particles in an accelerator. This includes designing the beamlines making up the accelerator, simulating the propagation of the beams, controlling active elements in the accelerator to improve performance, and interpreting the outputs from diagnostic devices. Furthermore, it is only the description of data used for these tasks, as contrasted with their physics and engineering, that is to be discussed.

Limited in this way, the subject is as much computer science as it is physics and engineering. Many, or even most, users of the accelerator programs look primarily for ease of use and are not particularly sympathetic to the issues discussed here. But, after working diligently for decades, providers of these codes are well aware of the difficulties. Most of the difficulties are shared by all computer-dominated tasks. Accelerator program developers, numbering in the hundreds, are facing much the same problems as the hundreds of thousands of program developers in the rest of the world. Naturally, to the extent possible they attempt to import up-to-date developments from the outside world into the accelerator domain. This is a constant source of tension however, because of the inevitable dependence by working facilities and control systems, on *ancient* (but largely satisfactory) software. Without repeating to this point, one should realize that the evolution of accelerator descriptions has been influenced by this tension.

Another source of tension has been due to the evolution from a small number of small groups to a large number of large groups, all working on similar problems. Already in a small group program *modularity* is important, so that different people can work simultaneously on different problems. The importance of *interfaces* is also present at this stage. But it is relatively easy and not particularly error-prone, when used by only a small group, for the interfaces to be informal. In this environment a single, all-purpose code, addressing a relatively small range of problems is very practical. As accelerators have become more complex and diverse, a more formal approach, taking advantage of concurrent advances in computer science, has become appropriate. Without these advances the ordinary user would by now be paralyzed by the complexity of the devices being described.

The following sections describe the gradual evolution from an all-purpose-program approach to a more formal, more modularized, more flexible environment in which many workers can both supply, and take advantage of, diverse tools without being unduly concerned about "bookkeeping" errors resulting from the sheer complexity of the systems.

MULTI-PURPOSE INPUT LANGUAGE

The early history of accelerator simulation consisted of diverse programs such as TRANSPORT, SYNCH, COMFORT, DIMAD, MAD, MARYLIE, and TEAPOT, for example as described by E. Keil[1]. These ``multipurpose" programs applied then, and in one form or another, still apply to a variety of projects. They accomplish a variety of tasks, such as design optimization, correction, tracking, and instability analysis. They use a variety of algorithms, such as nonlinear approaches, aberration formalism, Lie-algebraic techniques, symplectic integration, differential algebra; *etc.*

An important step in coordinating input descriptions for these efforts occurred in 1984 workshop[2] which suggested MAD input language[3] as Standard Input Format (SIF). The MAD/SIF major features were:

- Comprehensive accelerator model based on two major concepts: elements and beam lines
- Classification of element types
- Classification of their attributes
- Convention grammar, based of a single rule:
  label: keyword {,attribute }
- Directives triggering procedural program mechanisms such as subroutines, loops, variable assignment, and others.

Though this language has held up fairly well, the adaptation of the MAD parser to a different program is made difficult by its reliance on FORTRAN. This has led to the development of numerous "dialects". Also, a convenient set of MAD directives could not substitute for the power of the standard programming languages. As a result, it prevented user-specific extensions and the description of complex scenarios, such as tune modulation, etc.

APPLICATION PROGRAMMING INTERFACE

Further development of accelerator technologies and applications introduced new computational tasks associated with the study of new physical effects, devices and their complex combination. This tendency emphasized extensibility as one of the major criteria of the accelerator programs. In the previous section, it has been already mentioned that the approach based on the embedded parsers could not address all spectra of new requirements. At that time, an interesting and influential solution had been implemented by M. Berz in COSY INFINITY[4] which suggested procedures for constructing and adding new elements:

```
INCLUDE 'COSY'
PROCEDURE RUN;
  PROCEDURE SQ PHI L B D;
  . . .
  ENDPROCEDURE;
OV 5 2 0;
UM;
DL .1; {drift}
SQ 30 .2 .1 .1;
PM 6;
ENDPROCEDURE;
RUN;
END;
```

In the above example, the new element type SQ is added "on the fly" in the user program and processed together with the COSY INFINITY conventional drift DL. Such a dynamic mechanism provided accelerator physicists with a powerful tool for going beyond the standard descriptions and solving numerous differential algebra-based applications.

The ideas and development of object-oriented technology brought a new basis for revising and extending the previous approaches. The automatic differentiation suggested by L. Michelotti in the MXYZPTLK code [5] was the first important illustration of the powerful C++ concepts in the context of accelerator physics applications. Soon, this approach was extended in the PAC++ framework [6] for describing accelerator elements and beam lattices. In PAC++, the accelerator element was considered as superposition of MAD parameters:

```
SBend hb = length*L + 2*PI/N*ANGLE;
```

where L and ANGLE are global instances of the Attribute class, and length, PI, N – double variables. The assignment, addition and multiplication are implemented by overloaded operators that build the key-value associations of the element attributes. The new scripting languages allowed further simplification of this description with the help of the built-in containers. For example, in Perl, the same *hb* element can be described as:

```
$hb->set("l" => $length, "angle" => 2*$pi/$n);
```

The primary goal of this approach was to get rid of the existing embedded parser and to bring the full power of the standard programming language to users for describing complex accelerator scenarios and supporting new extensions. For the same reasons and to a much larger extent, the similar C++ approach is very popular in the high energy physics applications. For example, the following extract of the BNL STAR detector file [7] is described directly via the class methods of the new ROOT Geometry package [8]:

```
TGeoCombiTrans* ct_tpad31000 =
                              new TGeoCombiTrans();
tpss->AddNode(tpad3,1000,ct_tpad31000);
ct_tpad31000->RotateX(0.0);
ct_tpad31000->RotateY(0.0);
ct_tpad31000->RotateZ(15.0);
```

On the other hand, programming the lattice description in C++, Perl or other standard languages also has serious drawbacks, particularly, because of their strong bias towards the associated software environment. This issue becomes very important in modern multilab projects and will be considered in the next section.

EVOLUTION OF EXCHANGE FORMATS

Early Versions

Less ambitious than standardizing computational procedures is to standardize the description of the elements making up the accelerator, intentionally excluding any implication whatsoever as to how particles and beams will propagate through them. Already at this level it is sensible to distinguish between two levels of specificity. For the designer it is most valuable to have a compact description of a lattice of identical, ideal elements, with parameters conveniently expressed by algebraic formulas. But the "as-built" accelerator has non-identical elements, all of whose elements can be expressed as numbers. It is the latter form of lattice that is appropriate for control systems, and it is the form in which "save sets" (snapshots of all parameters valid at a fixed time) can be written. Such files can be referred to as “fully-instantiated”, and are sometimes referred to as “flat”, because of the relative simplicity of their database storage.

An example of full instantiation was the *writefile*, *readfile* pair of directives in TEAPOT[9] (1997). By 1998, in connection with the US-LHC collaboration, to support collaboration among workers at remote locations, the need had become clear for standardization of fully instantiated lattice descriptions. For simple lattices it is not difficult to repeat detailed orbit steering and retunings on every computer run, but for a complex lattice, “tuned-up”, fully-instantiated lattices need to be shared among remote designers. Suggested design principles for a fully-instantiated exchange protocol were spelled out in a 1998 letter[10]. As one response, the Standard eXchange Format (SXF[11]) was developed and (crucial to its successful use) was made routinely available from MAD-X. SXF has been used since then within Unified Accelerator Libraries (UAL).

ADXF 1.0

At the same time that SXF was being developed, an exchange format called Accelerator Description Exchange Format (ADXF[12]), based on the newly-popular computer language XML was also developed. Though “markup” is a term that is specific to the fields of type-setting and publication, this language has proved to be surprisingly appropriate for describing complicated datasets, which includes accelerator descriptions.

Some ADXF features, in addition to full-instantiation, and responsive to principles expressed in [10], are:

- It mimics SIF to the extent possible, retaining basic accelerator objects and their attributes.
- It represents the accelerator by a flattened tree of accelerator nodes, elements and sequences. Sequences can be nested to an arbitrary depth and may have references to the corresponding design beam lines they instantiate.
- It is *minimally complete*, meaning it describes all components that influence or monitor particle motion, and *only those*.
- It is *extensible*, meaning that it supports (but does not encourage frivolous) extension in two ways: introduction of new element types and introduction of new “element buckets” (parameter containers) common to all element types.
- Shared (for example between two rings) lines are supported.
- The description should be “consistent across different phases, from conceptual design, through engineering design and analysis, to operation.” Though ADXF provides only a flat “operational” view of the accelerator, it provides a mechanism for reconstruction of the idealized hierarchical model from which it descended.
- Containing only element and lattice descriptions, and no beam dynamics, ADXF is usable without prejudice by any physical method.
- The principles of “multiple-realization” and “compliance with computer standards” expressed in [10] are met by the adoption of XML.

ADXF 2.0

ADXF has been updated to version 2.0. The most substantial revision was motivated by the requirements of integrating E.Forest's PTC concept of “fibre bundle”[13] into the ADXF model. The essential features of this integration are illustrated in Fig. 1, which can be compared to Fig. 2. In both cases the core description is *tree* of elements and sequences (or sectors) of elements. (Roughly speaking) in ADXF2.0 this tree is broken and expanded in order to permit the assignment of positioning and orientation attributes to whole sectors which are then referred to as ``frames'' aliasing PTC “fibre”. In this way positioning attributes can be added to “on-the-bench” or “uninstalled” elements to produce “as-installed” elements.

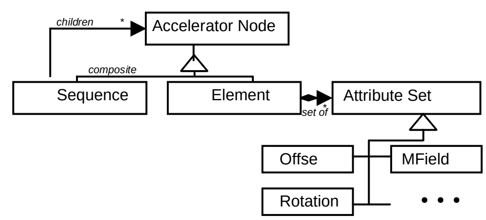

Figure 1: ADXF 1.0 model.

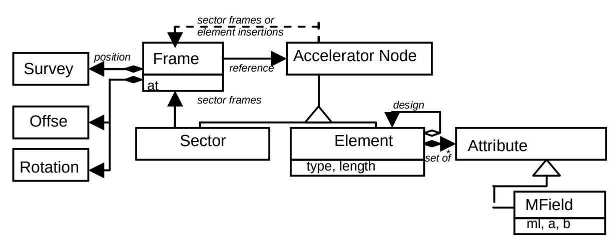

Figure 2: ADXF 2.0 model.

Though the 1.0 and 2.0 versions appear different in the figures, much of the apparent difference has come from the need for treating positioning and powering attributes differently. The instantiation of (intentional) “survey” positioning (possibly different for the same element in different lines) and (presumeably unintentional, but necessarily tied to a physical element) offsets and rotations are also exhibited in Fig.2. The ADXF2.0 model is built from five main ingredients:

- An *accelerator* is any sector selected by the user.

- A *sector* is a named sequence of frames with *installed* accelerator components.
- A *frame* is a layout of rigidly-associated installed components. It contains a relative position, misalignments, and a reference to an associated sector or accelerator element.
- An *accelerator element* is an accelerator device or positioned physical effect, such as a beam-beam interaction. All accelerator elements have the same structure: name, length, and an open collection of attribute sets. An element may also have a reference to its design version.
- An *element attribute set* is a container of attributes relevant to a single physical effect or feature, such as magnetic field, aperture, *etc*.

The concept of a *frame* as a layout of installed components addresses several accelerator applications. First, it facilitates the study of multi-line systems sharing the same sector. The list of such applications is extensive: injection and extraction systems, interaction regions, recirculators, and many others. Second, the new ADXF model becomes consistent with the detector description based on positioned volumes. For example, according to the ROOT Users' guide [14]:

> "The basic components used for building the logical hierarchy of the [detector] geometry are the positioned volumes called nodes. Volumes are fully defined geometrical objects having a given shape and medium and possible containing a list of nodes. Nodes represent just positioned instances of volumes inside a container volume".

Such similarity not only justifies the new accelerator model, but also facilitates the integration of accelerator and high energy physics software for modeling the cross-domain tasks, e.g. background study.

Extensibility of *accelerator element* types and *attribute sets* in ADXF 2.0 is provided by the consistent object-oriented mechanism of the XML schema, which eventually resolved the deficiencies of the previous DTD-based approach. For example, all MAD elements are implemented as descendents of the ADXF generic accelerator component, and we can use the conventional MAD terminology with the XML flavor:

```
<elements>
  <marker name="mk1" />
  <sbend name="bend" l="lq" angle="deltheta" />
  <quadrupole name="quadhv" l="lq" k1="kq1" />
  …
</elements>
```

In addition, in the boundary of the same schema, the MAD-oriented design description can be connected with another view or extended with SXF-like operational collection of generic elements:

```
<elements>
  <sbend name="d0mp08" l="3.58896"
                    angle="-0.0151186" />
  <element name="bi8-dh0" design="d0mp08">
    <mfield b="0 0 0.005476 0.033503"
          a="0. 0 -0.010166 0.024366" />
   </element>
   …
</elements>
```

where the <mfield> tag is associated with a set of measured magnetic field attributes of the bi8-dh0 element, designed after the MAD sbend d0mp08.

Another immediate practical advantage of the XML schema is its adherence to the collection of the schema-aware tools, editors or postprocessors. Such a schema-aware editor presents a full set of all legal options for every entry and is guaranteed to produce only schema-compliant lattice descriptions. As well as eliminating many sources of error, this facilitates the subsequent generation of other lattice descriptions. There are powerful XML tools for converting a schema-compliant file into other formats. Hence, for example, though it is difficult (and not necessary possible in principle) to convert a MAD file into and ADXF file, it is very easy and robust to convert an ADXF file (satisfying a MAD-specific schema) into a MAD file. For a given lattice, to apply a simulation code that requires proprietary input is therefore relatively straightforward once one has the lattice in ADXF form. Finally, intermediate, tuned-up lattices, typically fully-instantiated in numerical form, can be edited using the same tools as used on the original design lattice.

PROPAGATOR FORMAT EXTENSION

Multi-purpose accelerator program input files usually combine the accelerator element description along with directives defining the calculations to be performed. As discussed in the introduction, this organization is appropriate for small, relatively specialized tasks.

Rather than following this approach, to improve the code modularity, the Accelerator Propagator Framework was introduced into UAL [15]. This is an environment in which a variety of (independently-generated) tools can be applied to the same accelerator lattice. The first step was to adopt the Element-Algorithm-Probe conceptualization of accelerator simulation ingredients. *Elements* are magnets, RF cavities, etc. Their parameters are fully described in the ADXF file. All quantities whose evolution around the lattice are of interest and are mathematically-calculable are referred to as *probes*. Examples are individual particle coordinates, bunch centroid coordinates, transfer matrices, transfer maps (expressed as truncated power series), Twiss functions, and so on. Finally, *algorithms* are the mathematical formulas which implement propagation computationally. The APF object model is shown in Fig. 3, where the element and algorithm concepts are represented correspondingly by the Accelerator Node and Propagator Node.

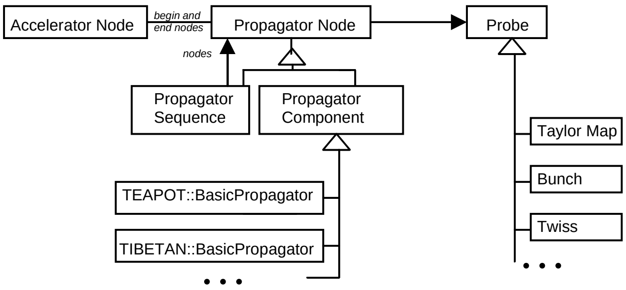

Figure 3: Accelerator Propagator Framework

With this organization, all that remains is to associate each element with the algorithms to be used to evolve all needed probes through the element. This is the function of a separate file called Accelerator Propagator Description Format (APDF). Since this file can be put into one-to-one correspondence with the full sequence of elements making up the lattice, it is, logically, a list of algorithm names, one for each element. The APDF file might therefore be expected to have roughly the same length as the ADXF file to which it is to be associated. In fact, since the same algorithms are applied to whole classes of elements, with natural defaults, the APDF file is typically very short. In addition, in APDF each propagator may be associated with an entire accelerator sector. This scheme allows one to bridge the gap between element-by-element and map-based approaches.

A sample APDF file (used in a recent emittance growth calculation) is:

```
<apdf>
<propagator id="stringsc" accelerator="ring">
<create>
<link algorithm="DriftStringSCKick" types="Default" />
 <link algorithm="DriftTracker" types="Marker" />
<link algorithm="DriftStringSCKick" types="Drift" />
<link algorithm="DipoleStringSCKick" types="Sbend"/>
<link algorithm="MltTracker"
  types="Quadrupole|Sextupole|Multipole|[VH]kicker"/>
 <link algorithm="RFCavityTracker" types="RfCavity"/>
 </create>
</propagator>
</apdf>
```

(To make this listing fit the column format of this report, the full designations of the classes to which these methods apply have been suppressed; the full class designation is, in fact, required.) This file controls evolution through an arbitrary lattice with intrabeam space charge forces taken into account. To turn off all space charge calculation (for example because it is too time-consuming) and revert to default tracking (which is thin element tracking) one need only remove the lines containing methods whose names include "StringSCKick" in this file. No changes are needed in the ADXF file. (It is philosophically satisfactory that a lattice description has no reason, in principle, to be aware of the algorithms to be used in simulating beam evolution through the lattice.)

Briefly, the attributes of the *links* listed in the APDF file are:

- A *sector* is a pair of begin and end accelerator element design names, e.g. d1, qf1 defining a sector that includes d1 but not qf1.
- An *element* is a regular expression that selects accelerator nodes by their name; e.g. q1|q2.
- A *type* is a regular expression that selects accelerator nodes by their type; e.g. Quadrupole|Sextupole.
- An *algorithm* is the full class name of the associated propagator; e.g. TEAPOT::MltTracker.

Sector, elements, and types define three different approaches for selecting families of accelerator elements. In case of overlapping, sector-based priority is first, element-based priority is second, and type-based priority is third.

APDF addresses a spectrum of applications ranging from small special tasks to full-scale, realistic models encompassing heterogeneous effects. Some modeling scenarios are indicated in Table 1.

Table 1: Some APDF-based applications

	Application	Configurable Propagator
1	Longitudinal beam dynamics	2D matrices + RF tracker
2	Linear lattice functions	4D tracker
3	Fast tracking	Combination of maps and thin elements
4	Instrumentation modeling, e.g. beam transfer function	#3 + propagators for active diagnostics devices, such as AC dipole
5	Dynamic aperture, halo, IR background investigation	element-type association
6	Special localized effect, beam-beam, impedance, ions	#3 or #5 + propagator for special effect
7	Post-processing analysis (e.g. MIA) and visualization	all of the above + post-processing aware virtual devices
8	Full-scale "realistic" model	All of the above

COMPOSITE APPROACH

The variety and evolution of different approaches suggested that an optimal program interface must be built as some their combination. The choice of the particular structure depends on many factors: legacy of accumulated programs, scope of the project applications, available resources, background and preferences of developers, and many others. According to our experience with the development of Unified Accelerator Libraries (UAL) environment, we recognize the following three approaches:

- Application Programming Interface
- Lattice Exchange Format
- Propagator Description Format

In UAL, the user interface is implemented by the C++ class Shell serving as a project-oriented façade to underlying structures and algorithms. The user application starts with reading the exchange format generated from the MAD-X design toolkit, online model or defined by user. To address both legacy and new applications, UAL supports two lattice exchange formats: SXF and ADXF 2.0. After the initialization of the accelerator containers, the user can access and directly update them with the C++ interface, for example, for distributing random errors, simulating tune modulation, and so on. Within the Model Player interactive analysis environment [16] the Accelerator Propagator Description Format (APDF) has become a necessary part of all present RHIC offline and online applications.

REFERENCES

[1] E. Keil, "Computer Programs in Accelerator Physics," AIP, 1985

[2] Workshop for the Standardization of MAD Input Language for Beam Optics, Stanford, 1984

[3] D.C. Carey and F.C. Iselin. "Standard Input Language for Particle Beam and Accelerator Computer Programs," Snowmass, Colorado, 1984

[4] M. Berz, "Computational aspects of design and simulation: COSY INFINITY." Nuclear Instruments and Methods, A298:473, 1990

[5] L. Michelotti, "MXYZPTLK: A practical, user-friendly C++ implementation of differential algebra: Users's guide," FN-535, 1990

[6] N. Malitsky, A. Reshetov, and G. Bourianoff, "PAC++: Object-Oriented Platform for Accelerator Codes," SSCL-675, 1994

[7] STAR team, private communication, 2006

[8] A. Gheata, "GEOM: Status and developments," ROOT Workshop, 2005

[9] L. Schachinger and R. Talman, "Teapot: A Thin-Element Accelerator Program for Optics and Tracking," Particle Accelerators, 22, 35(1987)

[10] F.C. selin, E. Keil, R. Talman. Letter to ICFA Beam Dynamics Newsletter, 21 January, 1998

[11] F. Pilat et.al., "Standard eXchange Format (SXF) for Accelerator Description." RHIC/AP/155, 1998

[12] N. Malitsky and R. Talman "Accelerator Description Exchange Format," ICAP, 1998

[13] E. Forest et al., "Polymorphic Tracking Code PTC," KEK Report 2002-3.

[14] R. Brun et al, "ROOT User's Guide," http://root.cern.ch

[15] N. Malitsky and R. Talman, "The Framework of Unified Accelerator Libraries," ICAP, 1998, http://www.ual.bnl.gov

[16] V. Fine, N. Malitsky, R. Talman, "Interactive analysis environment of Unified Accelerator Libraries," Nuclear Instruments and Methods, A 559, 2006