EXPERIENCE WITH THE CLASSIC LIBRARY IN MAD VERSION 9

F. Christoph Iselin*
CERN, SL/AP group, CH-1211 Geneva, Switzerland

Abstract

The CLASSIC library is a C++ class library which provides services for building portable accelerator models and algorithms for their analysis. This paper describes the motivations behind the CLASSIC library and its main features. It shows how this library can be used in a large accelerator design program like the new version 9 of MAD written in C++. The possibilities are illustrated by presenting some new developments in MAD version 9, like sophisticated matching features with simultaneous matching of two rings.

The major part of the CLASSIC library is now implemented. Its source code and some preliminary documentation are available from the author.

1 INTRODUCTION

The complete design of an accelerator usually requires many different problems to be solved, and to do so, several programs to be run in succession. This implies that accelerator description data must to be sent back and forth between programs, which is complex and error prone due to unnecessary differences in input formats. Data exchange could be avoided if one could pull an algorithm out of one program and plug it into another. However, most of the time this is impossible, since the internal data structures vary wildly between programs.

The format problem was partially solved in 1984, when a "Standard Input Language" was proposed. Nevertheless the data structure problem remained with all problems it causes. To alleviate this problems John Irwin (SLAC) proposed the development of a C++ class library called CLASSIC (Class Library for Accelerator System SImulation and Control). The project was started during a workshop held in summer 1995 at SLAC as an international collaboration.

Section 2 summarises the history of the CLASSIC project. Section 3 describes the features and use of the CLASSIC library, and Section 4 outlines a few new features in MAD which became possible thanks to the CLASSIC library.

2 HISTORY OF CLASSIC

The CLASSIC library was initially started by a large collaboration. Many persons promised to participate, but many of them have not been able to keep their promises due to other commitments. The most important contributions were made by (in alphabetical order)

Scott Berg (SLAC/CERN/Indiana University),

Yunhai Cai (SLAC),

Alex Dragt (University of Maryland),

James Holt (FNAL),

John Irwin (SLAC),

Christoph Iselin (CERN),

Leo Michelotti (FNAL),

Nicolas Walker (DESY),

Yiton Yan (SLAC),

Johannes van Zeijts (TJNAF).

During the years 1995 and 1996 these persons met in several workshops to define a structure for the CLASSIC class library. The result was first presented in 1996 [2]. Since then most of the library has been implemented, and is now available from the author.

3 DESCRIPTION OF CLASSIC

3.1 Terminology

Throughout this description the following terms are used:

Component: A single object occurring in an accelerator, like a magnet or a drift space.

Beam line: A sequence of accelerator elements. Beam lines can be nested to any depth.

Element: Anything that can occur in an accelerator. This can be a single component, or an arbitrary sequence of components.

3.2 Structure of CLASSIC

The CLASSIC library consists of several groups of classes called class categories. Each class category provides the user with a logically distinct service. 14 class categories are now completed:

AbsBeamline: This category contains a set of abstract classes which define the interface of single components, like drift spaces, magnets, electrostatic elements, RF cavities, etc. The implementations can be found in the category `BeamlineCore`.

* email: chris.iselin@cern.ch

BeamlineCore: This category contains the classes providing concrete representations of components. The instance objects of these classes contain all data needed for performing computations on these components. This category could easily be replaced by another one which provides a different representation. For example, a representation might implement a direct interface to a data base. The separation between `AbstractBeamline` and `BeamlineCore` makes it possible to decouple algorithms completely from component classes.

ComponentWrappers: Each "Wrapper" component inherits its interface from the corresponding abstract element. It uses the "Decorator" pattern [3] to attach field changes, like random field imperfections and/or field scaling to a component.

Beamlines: The template classes in this category represent the structure of beam lines. Some of them also allow to store arbitrary data in all positions in a structure. In particular they provide mechanisms for attaching misalignments to any element (be it a component of a beam line). They also permit to build tables such as tables of optical functions or of survey data.

BeamlineGeometry: This category contains representations for all different geometries which may occur in a component or in a beam line. Each element (component or beam line) contains a geometry object, which defines the design orbit of the accelerator. Classes exist which model geometries comprising Cartesian coordinates (straight geometry), mid-plane geometry (sector bend like), and many more.

Fields: This category includes all magnetic and electric field types which can exist in components (dipole, multipole, RF field, etc.).

Construction: This category implements the "Factory" pattern [3]. A `ElementFactory` object can construct components and beam lines. It can be called directly by the user, but it is also used in the language parser for constructing components.

Channels: A `Channel` implements the "Proxy" pattern [3]. It controls access to arbitrary attributes of components, and may be used in a matching process to provide uniform access to all matching variables and constraints.

Algorithms: The `Algorithm` classes implement the "Visitor" pattern [3]. A "Visitor" can be thought of as an algorithm object which walks through a beam line and applies itself to each component in turn. Presently `Algorithms` includes a `Surveyor`, which computes the global geometry, and several mapper algorithms which accumulate linear or non-linear transfer maps using various algorithms. `Algorithms` also contains a few special integrators which can be attached to a component or beam line.

Algebra: Here one finds templates for abstract data types like `Vector`, `Matrix`, `Tps` (Truncated Power Series), `Vps` (Vector Power Series), and maps, all with variable dimensions. Their template parameter selects the data type from which they are built (`int`, `float`, `double`, or even `Tps`.). There are also methods for finding fixed points and non-resonant normal form analysis. Further algorithms shall be added in future.

MemoryManagement: A pair of classes implementing reference-counted shared objects.

Parser: This category contains a simple parser for the Standard Input Language (SIL) [1]. It constructs an accelerator structure from a SIL input file. For this purpose it uses the class `ElementFactory` from class category `Construction`.

Physics: The class `Physics` defines a name space defining mathematical and physical constants.

Utilities: This category contains a random generator as described by Knuth [5] and a collection of exception objects. The exception objects are used throughout the other categories to handle errors.

3.3 Use of CLASSIC

First, an accelerator structure must be built using the parser included with the library, or by defining accelerator components as `C++` objects and combining them by executing a `C++` program. The resulting structure may contain beam lines and sub-lines nested to arbitrary depth. Within the structure the magnetic and electric fields are represented by classes all derived from a single base class. This permits to treat them all uniformly. The geometry of each accelerator element is described by its contained geometry object.

Next, the user may generate random or systematic misalignment errors for selected elements (components or whole beam lines). Selected components can have field modifiers (random or systematic errors, or scaling) attached.

Special integrators can be defined for any element. When it makes sense for a given algorithm, such an integrator replaces the normal function of the algorithm. An integrator could e. g. use a precomputed map for a complex component or for a complete beam line. Another integrator example automatically splits a component into several thin lenses.

Once the structure is built, the user can apply an algorithm, a `C++` object which encapsulates the physics computations to be done. An algorithm can perform any computation on the beam line, such as tabulation of lattice functions. An algorithm is sent to a beam line as a "Visitor" pattern [3]. Both the beam line representation and the algorithm are implemented by abstract classes. This efficiently

hides the implementation details and greatly reduces coupling between classes.

Matching of an accelerator structure is made simple by `Channel` objects. Once defined, these allow uniform direct access to arbitrary attributes of components. By building an array of channels for access to the variable parameters, and another array of channels for access to the constrained values, the matching process becomes a simple algebraic problem, like

```
// Define an algorithm object.
Mapper visitor(...);
Vector<Channel>variables;
Vector<Channel>constraints;
// Fill in the channels for
// variables and constraints.
...
// Assign new values to the variables.
variables = ...;
// Compute the new constraints.
visitor.execute();
// Extract the new constraint values.
... = constraints;
```

This example is by no means complete, but it should give the flavour of the method.

3.4 State of CLASSIC

Since CLASSIC was presented in 1996 as a project [2], the major parts of the library have been completed. The library has been used successfully in a new version of the MAD program, and many minor changes were made to make its use easier. The library compiles cleanly with the latest version of the `egcs` compiler (1.1b). It does however not yet compile with a purely standard-conforming compiler, since for the namespace features have been omitted. This point will be corrected soon. The CLASSIC source code is available from the author [6] under the same conditions as the CERN program library.

Even though the majority of the library is now complete, some important algorithms, as well as the planned interface to the control system are still missing. Development goes onto fill these gaps.

For the time being the documentation uses `doc++`-like comments [4]. The resulting documentation is available on the world-wide web [7].

3.5 Timing Benchmarks

Most of the CPU time is usually spent for handling truncated power series. Benchmarks have shown that for moderately high orders (about 6) the methods of CLASSIC are comparable in CPU time usage to Berz's FORTRAN routines [8].

For linear maps it turns out that a large fraction of the CPU time is used for memory allocation and deallocation, but studies are under way to overcome this problem. A linear algebra package and a differential algebra package with fixed dimensions will possibly be added for speed.

3.6 Future Plans

In near future, many more algorithms will be defined and added to the library. Some of these will be lifted from MAD, once they are sufficiently tested.

Normal form analysis is now only possible in the non-resonant case. The resonant case will be added shortly.

The CLASSIC structure is now more or less frozen. The documentation can thus be rewritten in a more digestible form. It will include a user's guide and a reference manual.

4 ADVANCED FEATURES IN MAD

The CLASSIC library made it possible to implement several advanced features in the new `C++` version of MAD. These features are mainly aimed at the design of a collider with two separate rings with two-in-one magnets like the LHC machine, for which many particular problems arise.

- The machine has a global coordinate system, whose arc length s coincides for both rings, even though one of the beam runs in opposite direction towards decreasing s.
- Many components in an interaction region are common to both rings. Obviously for physical consistency their order must agree for both rings. This suggests that they are defined by the same sequence definition. The common parts are however traversed in opposite sense by the two beams. This changes the sign of the magnetic forces.
- The layout of the two rings are strongly coupled to each other due to the two-in-one design of the magnets.
- The twin-bore magnets cause strong correlations between the field imperfection in either aperture.
- The small apertures and the lack of space impose very tight matching conditions.
- For the LHC machine, the two rings must fit the existing LEP tunnel. Many present-day accelerators are constrained by similar types of constraints.
- When matching parts of the machine, e. g. interaction regions or arc cells, all algorithms should see consistent imperfections whenever the same accelerator components are seen in different operations using different ranges of the machine.
- Interaction regions must be adjusted to match arc cells at either end, the most efficient method requires backward tracking of lattice functions in the downstream cells. This implies sign changes in the algorithms, which are different from those induced by a beam running in backward direction.

- In order to fulfil all constraints in an optimal way, it is desirable to match both rings in the same time, together with their geometric layout.

4.1 Optical Tables

For efficient matching MAD-9 implements the concept of tables. Their interaction with other objects is shown schematically in Fig. 1. These tables can be used for matching as described below.

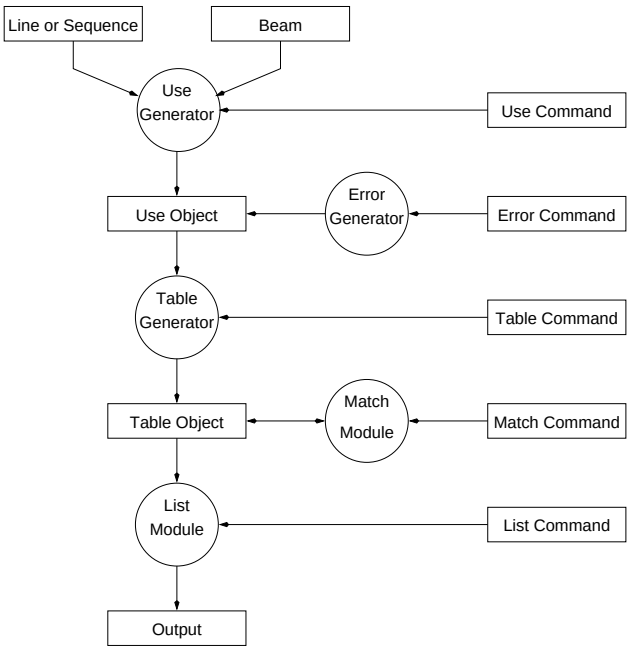

Figure 1: Schematic view of the Interaction between Objects in MAD

The use generator creates a use object; this is a fully instantiated copy of the input beam line, where all components are made unique. The use object also has an attached beam, defining the particles running through the line and their reference energy. The error generator generates misalignment and multipole field errors as desired and attaches them to the use object. The table generator pulls out a flat representation of a selected range of the use object and fills the attached data slots with data such as lattice functions or survey data. The match module interacts with one or more table objects and adjusts their parameters to obey the matching constraints. Finally the list module extracts a formated listing from the table object.

At the time of writing the following tables are available:

- Periodic linear lattice functions.
- Linear lattice functions with initial values.
- Geometric layout.

A table containing the transfer maps accumulated from its beginning to the current position is being implemented.

4.2 Matching

In a matching process all global variables and all object attributes can be made variable.

Matching constraints can be arbitrary expressions containing:

- Object attributes (e. g. the field strength in a magnet),
- Global Variables (e. g. the global reference momentum),
- Summary values from a table (e. g. the machine tune),
- Entries from tables (e. g. the β_x value in a selected position),
- Vectors built from the above (e. g. the r.m.s. value of dispersion over a selected range in a table).

Very general matching constraints can be constructed. Several tables can be used in the same matching process. By default they are all recomputed whenever a parameter changes, such as to keep all constraints up to date. If a table is known to remain constant, it can be declared as static. It is then not recomputed to save CPU time.

4.3 Status of MAD-9

A beta test version is completed. It is being used to test the early stages of LHC version 6. Its features include:

- Input of the machine structure in a much more flexible way than in MAD version 8.
- Definition of misalignments and/or field imperfections. The program now accepts all order of multipoles, even for elements which are by definition single multipoles.
- Computation of linear lattice functions.
- Computation of geometric layout.
- Computation of arbitrary order maps with normal form analysis. So far only the non-resonant case can be handled.
- Matching of lattice functions or layout.
- Simultaneous matching of lattice functions and layout.
- Simple tracking of particles.
- Interaction with the DOOM data base [10].

4.4 Future Plans for MAD

Plans for near future include:

- The beam line structures of CLASSIC are set up such as to make it easy to run through a beam line in either direction. The `Visitor` pattern encapsulates the complete state of an algorithm. This makes it easy to keep track of the proper sign changes for all four interesting cases:

  1. Run beam forward and track in the same direction.
  2. Run beam backward and track in the same direction.
  3. Run beam forward and track in the opposite direction.
  4. Run beam backward and track in the opposite direction.

  The case (1) is complete, the three other cases are being implemented.

- Beams running in reverse direction through a lattice.
- Tracking of particles or lattice functions in the same direction or in the direction opposite to the beam.
- Tables of accumulated maps.
- The documentation [11] must be completed.

5 CONCLUSION

It has been shown that the CLASSIC library design is viable, and that it provides the flexibility needed for implementing very sophisticated features in an accelerator design program. Some research is still needed to make the library more efficient, and more algorithms should be implemented based on CLASSIC. The control system interface, as well as a data base interface should be defined and written. For the latter the CORBA specification might be helpful.

6 REFERENCES

[1] D. C. Carey and F. C. Iselin: 'A Standard Input Language for Particle Beam and Accelerator Computer Programs'. 1984 Snowmass Summer Study.

[2] F. C. Iselin: 'The CLASSIC Project'. Computational Accelerator Physics Conference 1996, Williamsburg, VA.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, 'Design Patterns', New York, Addison-Wesley 1995.

[4] Malte Zöckler, Roland Wunderling: 'DOC++'. URL: `http://www.zib.de/Visual/software/doc++/index.html`.

[5] D. Knuth: 'The Art of Computer Programming', Vol. 2 'Semi-numerical Algorithms', 2nd edition, Addison-Wesley, 1973.

[6] F. C. Iselin: 'Classic source code'. URL: `http://wwwslap.cern.ch/~fci/classic/2.0`.

[7] F. C. Iselin: 'Classic Reference'. URL: `http://wwwslap.cern.ch/~fci/classic/doc`.

[8] M. Berz: 'Differential Algebra Package'. FORTRAN package available from: Department of Physics and Astronomy and National Super-conducting Cyclotron Laboratory, Michigan State University, MI.

[9] J. Chen, W. Akers, G. Heyes, D. Wu, and C. Watson: *An Object-Oriented Class Library for Developing Device Control Application, Proceedings ICALEPCS 1995*.

[10] H. Grote: 'The DOOM Project'. URL: `http://wwwslap.cern.ch/~hansg/doom/ap_doom.html`.

[11] F. C. Iselin: 'The MAD Program, Version 9.01 (Methodical Accelerator Design) User's Reference Manual'. URL: `http://wwwslap.cern.ch/~fci/mad/mad9/user_guide.html`.