ADVANCED MULTI-PROGRAM GUI FOR ACCELERATOR MODELING

T. J. Roberts, Muons, Inc., Batavia IL 60510, U.S.A.
D. M. Kaplan, Illinois Institute of Technology, Chicago IL 60616, U.S.A.

Abstract

There are dozens of programs for designing and modeling accelerator systems, most of which have their own language for describing the system. This means a designer must spend considerable time learning the languages of different programs and converting system descriptions among them. This paper describes a project to develop a new language for accelerator modeling, together with a portable suite of programs to implement it. These programs will assist the user while editing, visualizing, developing, simulating, and sharing models of accelerator components and systems. This suite is based on a Graphical User Interface (GUI) that will permit users to assemble their system graphically and then display it and check its sanity visually, even while using modeling programs that have no graphical or visualization capabilities. Incorporating the concept of libraries as a primary component of the language will encourage collaboration among geographically diverse teams. The requirements for developing this language and its tools will be based on generality, flexibility, extensibility, portability, usability, and sharability.

THE KEY CONCEPT IS COMMONALITY

All programs that model or design accelerator systems have considerable *commonality*, in that they all model accelerator systems, inherently using the same basic beamline components (e.g. quadrupole magnets, bending magnets, RF cavities, etc.). Implementing a dozen or so types of components, and permitting them to be placed sequentially on a beam centerline (with optional offsets and rotations), can accommodate the great majority of modern accelerator systems (except cyclotrons and other systems that have no lattice and cannot be handled by lattice design programs). A description of a system in terms of its components will be usable by any modeling program, as long as the tools exist to convert the original description into the input language of the modeling program. Rather than relying on language translation, which is a challenging problem even for very simple languages, the approach we use is *instantiation*. That is, a human developer specifies how each component is described in the input deck of each supported modeling program. So a system described as a series of components can be instantiated for any supported modeling program, automatically providing the input deck to run it. The method for doing this is so simple and general that knowledgeable users can implement the required instantiations for new components, and for new or unsupported modeling programs.

INTRODUCING LINGUAFRANCA

The working title of this new description language is "LinguaFranca", with obvious linguistic and historical roots. It is designed with the following general requirements in mind:

Generality accommodate all types of accelerator and beamline components; support as many lattice design and simulation programs as possible.

Flexibility make it easy for users to move components around, add or delete components, specify components' attributes, combine subsystems into systems, etc.

Extensibility permit users to define their own components and subsystems; permit users to add new modeling and simulation programs.

Portability programs must run on most operating system environments, specifically Linux, Windows, and Mac OS X.

Usability implement a modern GUI, but still permit users to easily edit their system descriptions using their favorite text editor.

Sharability make it easy to share components and system descriptions among disparate users via the web; components should be self-documenting.

The primary advantages of this approach are:

- Reduced learning curves for designers to use unfamiliar modeling programs.
- Greatly improved ability for designers to select the right modeling program for the job.
- For a given system, multiple modeling programs' results can be compared and contrasted much more easily.
- Designers can use graphical tools to construct and visualize the system, even when using modeling programs that don't support it.
- Components are self-documenting and can be published in libraries, fostering collaboration among users and groups via the Internet.
- Considerably reduced development effort compared to either translating description languages or implementing these graphical and comparison capabilities multiple times for the individual simulation programs.

STRAW-MAN LANGUAGE EXAMPLE

The design of LinguaFranca will begin with a survey of lattice design and modeling programs, and an analysis of their commonalities and differences. This has not yet begun, but we do have a straw-man language design

based on the syntax of G4beamline [1]. Commands start with '%' to distinguish them from commands for modeling programs. The LinguaFranca programs know nothing about any accelerator components; essentially the only thing the programs know is that components are defined in the input file or in libraries and are placed sequentially along the centerline of the beamline (with optional offsets and rotations). This gives users the ability to define any sort of new component, subject only to the constraint that the simulation program(s) can implement it. The behavior of each object is determined by the simulation programs, not by LinguaFranca. It certainly is possible to define a component known to some simulation programs but not to others, so a system using such a component can only be simulated by the set of programs that implement it – the tools will detect this and warn the user when instantiation is attempted for a program that cannot implement it (perhaps failing the instantiation altogether).

Rather than presenting a formal description of this straw-man language, a simple example is presented, with a few comments. Some general notes about this straw-man language:

- Attributes have double-precision real values unless specified as type=string.
- Units conversion factors are implicitly defined, including: mm, cm, m, km, tesla, gauss, kilogauss, MeV, etc.
- Real numbers can have units appended, so 1000mm is equal to 100cm and to 1.000m; users can use units different from those used by the various simulation programs, because units conversion happens automatically.
- The default units given in %attribute are used if and only if no unit is contained in the value when the attribute is specified.
- A \ at the end of a line joins the next line, removing the \ and the "newline" character. This occurs during LinguaFranca input processing and is not passed to any simulation program.
- Comments are as in C++ (// comment to end of line, /* block comment */). They are removed during LinguaFranca processing, and are not passed to any simulation program.
- The %instantiate command introduces one or more lines to emit when instantiating for the specified simulation program; the optional %once and %every commands (within a %instantiate) introduce lines to emit just for the first placement, and for every placement.
- In lines within %instantiate (to be output to the input file of a specific simulation program), all real expressions are evaluated to numbers, and all string-valued attributes have {attribute-name} replaced by their value.

This example illustrates the structure to describe a simple quadrupole that can be used by G4beamline, Transport, and MAD accelerator simulation codes. The quadrupole's description is wholly contained between the "%component" and "%endcomponent" commands. Properties of the quadrupole are described by the "%attribute" commands. Each accelerator program has its own description of the quadrupole implemented via the "%instantiate" command. Lastly, an HTML description is given after the "%description" command, which provides a human-readable description of the component.

Example – a generic component: SimpleQuadrupole

```
%component SimpleQuadrupole
%synopsis A simple quadrupole magnet, no fringe.
%attribute name type=string
%attribute length units=mm
%attribute aperture units=mm
%attribute gradient units=tesla/m
%attribute outerRadius units=mm
%icon gradient>0 ? "focus.png" : "defocus.png"
%instantiate G4beamline
%once
genericquad {name} length=length/mm \
   apertureRadius=aperture/mm \
   ironRadius=outerRadius/mm \
   gradient=gradient/(tesla/meter) \
   fringe=0 \
   openAperture=1
%every
place {name}
%instantiate transport
5. length/m gradient/kilogauss*aperture/m \
  aperture/cm '{name}' ;
%instantiate mad
{name}: QUAD, L=length/m, \
   GRAD=gradient/(kilogauss/m);
%description
<html>
... HTML description in detail, with optional images.
... This can be very long, so it normally comes last.
</html>
%endcomponent
```

In keeping with good design practices, LinguaFranca is inherently object oriented – the objects are components of beamlines. So a quadrupole magnet is a single component; if it is to be modeled with fringe fields, then it is instantiated with multiple lines for programs that separate the fringe fields from the quadrupole in their input decks. It should be clear that this approach is extremely flexible and general, and corresponds well to the way components are placed into beamlines, and to the way designers think about beamlines. Even though the attributes and units of a component are not those used by the modeling programs, expressions can be used to do the appropriate conversion(s) – in the example the *SimpleQuadrupole* has attributes *aperture* and *gradient*, which are converted to pole-tip field and aperture for Transport, in the proper units.

In this straw-man language, a system description file consists of a sequence of %component definitions and %place commands to insert the components into the

simulated world in the desired configuration. Components defined in a library can simply be referenced as *LibraryName:component*, making it trivial to combine components from multiple libraries. The tools will make it easy to connect to multiple libraries simultaneously, needing only a name and URL for each; as LinguaFranca development evolves, the distribution will probably include a list of known libraries and their URLs. Much of the functionality of LinguaFranca will be in its libraries; the development team will publish a library containing the common accelerator components that are instantiable into all supported simulation programs.

STRAW-MAN LINGUAFRANCA GRAPHICAL EDITOR

The graphical editor will be written in Java [3]. This gives portability across operating systems, and means that advanced tools and libraries can be applied to improve the efficiency and speed of software development.

The basic graphical editing concept is for the user to connect to any desired libraries, and simply drag the necessary components from a library into position in the system being edited. Connected libraries each appear as a window containing their contents, and can contain both individual components and subsystems. Users can also define and use components locally, without putting them into a library.

As software development proceeds, new modules will be added, to do such things as: tuning and optimizing a beamline, coordinating multiple programs (e.g. tune via Transport, then evaluate via G4beamline), generating and displaying beam profiles and plots from output files generated by simulation programs, plus anything else that seems appropriate. The intent is to transform the graphical editor into an Integrated Development Environment that streamlines the design and evaluation of accelerator systems.

Figure 1 shows an example of how the graphical editor might look after inserting a quadrupole triplet. Components are represented as icons; for some components like *SimpleQuadrupole*, the icon used depends on the values of attributes.

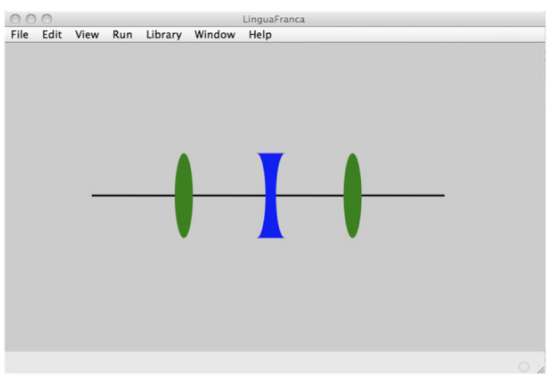

Figure 1 : Artist's conception of the Lingua Franca graphical editor. The three quadrupoles and four drifts were dragged from a library into the system, and their attributes were then edited.

LIBRARIES AND COLLABORATION

Beyond the LinguaFranca base library, it is intended that individual designers, groups, or laboratories publish libraries of their own, providing models of the magnets and other components in their inventory. The design of the LinguaFranca libraries will make it trivial for designers to publish and use libraries containing both subsystem designs and individual components. Self-documenting components are an important aspect of this, avoiding the separation of design data from human-readable descriptions, and permitting complete integration into the tools. This is intended to foster a new paradigm for accelerator design, modeled after the way modern software is developed, and facilitated by a set of tools that treat such libraries as inherent elements of the language. Rather than just sharing the tools and concepts, accelerator designers will be able to share the results of their design efforts in a simple and flexible way.

For instance, the design effort for a large new facility (such as a muon collider or a linear collider) could be split into numerous teams each working on one subsystem of the overall facility. Each team could publish its own library containing the components it is using in its designs; they could also publish their latest complete subsystem design in the library. An overall team could use the subsystem designs from multiple teams' libraries to verify the matching and interfaces between subsystems; potentially they could combine them all into an end-to-end simulation (although in some cases computational feasibility may prevent this from being useful). It is advantageous to be able to do all this using multiple modeling and simulation programs as checks on each other.

SUMMARY

LinguaFranca will give accelerator designers modern graphical tools and the ability to work with multiple modeling and simulation programs much more easily. This, in turn, will assist in the design of large systems by large teams of experts who are geographically dispersed and who may prefer different simulation programs for different parts of the system. By making it easy to compare and contrast the results from multiple simulation programs, this approach can potentially improve the accuracy of such designs, and the confidence in their accuracy and realism. The inclusion of libraries as first-class language elements will facilitate collaboration in new ways, and will open a new paradigm for accelerator design, using libraries of components and subsystem designs in new designs and systems, rather than starting essentially from scratch each time.

REFERENCES

[1] G4beamline: http://g4beamline.muonsinc.com
[2] Geant4: http://geant4.cern.ch
[3] Java: http://java.sun.com