Multiscale, multiphysics beam dynamics framework design and applications

James F Amundson[1], Douglas Dechow[2], Lois McInnes[3], Boyana Norris[3], Panagiotis Spentzouris[1] and Peter Stoltz[2]

[1] Fermi National Accelerator Laboratory, Batavia, IL 60510, USA
[2] Tech-X Corporation, Boulder, CO 80303, USA
[3] Argonne National Laboratory, Argonne, IL 60439, USA

E-mail: `amundson@fnal.gov`

Abstract. Modern beam dynamics simulations require nontrivial implementations of multiple physics models. We discuss how component framework design in combination with the Common Component Architecture's component model and implementation eases the process of incorporation of existing state-of-the-art models with newly-developed models. We discuss current developments in componentized beam dynamics software, emphasizing design issues and distribution issues.

1. Introduction

Accelerator-based science is a cornerstone of modern fundamental scientific research. As accelerator-based experiments have progressed, they have required better and better accelerators. "Better" can have several meanings, higher energy and higher luminosity being the two most common. Simulations are important tools for the design of future accelerators, as well as the the optimization of currently running accelerators.

A decade ago, state-of-the-art beam dynamics simulations were highly specialized. One program might have a detailed implementation of higher-order optics but lack or have only trivial implementations of collective effects such as space charge. Another program would implement a fully three-dimensional realization of space charge, but include only a trivial optics component. During the period of SciDAC-1, programs such as MaryLie/IMPACT (ML/I) [1] and Synergia [2] combined state-of-the-art optics codes with IMPACT, a fully three-dimensional, parallel space charge implementation. For the first time, beam dynamics simulations were able to combine detailed space-charge calculations with advanced, nonlinear optics.

While the approach of straightforwardly combining extant codes advanced the capabilities of beam dynamics simulations by enabling studies of the interplay between space charge and non-linear optics effects, extending the same approach to other processes and implementations would be difficult. During SciDAC-2, as part of the Community Petascale Project for Accelerator Science and Simulation (COMPASS) project, we have embarked on the next phase of beam simulation development by transforming our approach into a component-centered framework.

In these proceedings, we discuss the motivations for the component approach and the Common Component Architecture (CCA) implementation of component infrastructure. We then discuss some of the most important issues in component design for beam dynamics. Further,

we discuss a solution for the problem of distributing, porting, and building componentized applications.

2. Motivations for the component approach

In this paper we use the word “component” both in the colloquial sense as a well-defined constituent part and in the technical sense as defined by the Common Component Architecture Forum, as discussed in Section 4. For the moment, we restrict ourselves to the general sense of components of being portions of software that can be added and removed from multiple applications. For example, a space charge component is distinct from an application that merely implements space charge in a such a way that it cannot easily be separated from the remainder of the application’s implementation.

One goal of the component approach is to increase the availability of physics model implementations for use in simulations. Increased availability arises not only because components written for one application are available for others but also because the existence of well-defined interfaces and other infrastructure makes writing components much easier than it would otherwise be.

Availability of components enables previously impossible scientific applications, namely those that investigate the interplay between multiple physics processes (multiphysics). There are, however, other important advantages of the component approach. Different implementation strategies of the same physical effect allow us to choose the most efficient computational approach for the task at hand. For example, in beam dynamics problems where space charge effects are large, it is important to use a fully three-dimensional, self-consistent space charge implementation. In cases where the effects are relatively small, it may be possible to use a simpler two-dimensional approximation with substantially reduced computational cost.

Another important advantage of the component approach is the improved possibilities for validation and benchmarking. To date, most comparative benchmarks of collective beam dynamics effects have involved comparing entire codes against each other. The different codes typically have independent implementations of accelerator lattice construction and optics, beam generation, and so forth. Benchmarking two or more such codes in their entirety inevitably obscures small, but possibly important, deviations in the various aspects of the code. With component-based simulations, we will be able to perform benchmarks with only a single variable, namely, the component in question.

3. Synergia2 as a component framework

Synergia is a beam dynamics framework that is evolving into a fully component-based simulation tool. The original Synergia [2] was a hybrid code connecting IMPACT, a Fortran 90 three-dimensional space charge application with CHEF, a C++ single-particle beam dynamics library. The user interface for Synergia was controlled by Python, but the Python driver was limited to writing a set of input files to be read by the Fortran 90 application. This approach was successful in allowing the combination of nonlinear optics with space charge, but proved too difficult to extend to other processes.

Synergia has been replaced by Synergia2, a fully Python-driven framework capable of calling the CHEF libraries and the space charge portions of IMPACT as well as a new C++ space charge implementation, Sphyraena, and other modules. Synergia2 was designed for extensibility from the outset. The various modules are all considered on equal footing; new modules are in development. Synergia2 is rapidly becoming a component framework in the colloquial sense described in the previous section. All that is standing in the way for this level of componentization is the formalization of the component interfaces. We discuss some of the relevant design issues in Section 5.

There are still several challenges faced by the Synergia2 componentization effort. The foremost challenge is the difficulty of dealing with cross-language issues. Synergia2 is a Python application with components written in Python, C++, and Fortran 90. Although we have managed to overcome the C++-Python interface issues with little difficulty, the interface to Fortran 90 has proved more problematic. Our current Fortran 90 component implementation was difficult to write, is somewhat fragile, and suffers from portability problems. Furthermore, if other (non-Python) applications want to take advantage of Synergia2 components, they have to deal with their own set of language interoperability problems. We plan to solve these problems with the CCA tools described in the following section.

4. Componentization using the Common Component Architecture

Rapid advances and an increasing diversity in high-performance computing platforms have placed technical computing burdens on the shoulders of scientists who use, as opposed to study, computation. This situation motivates scientists to take advantage of advances in computer science designed to reduce these burdens. Language interoperability is one such problem we have discussed in the context of Synergia2, but many others loom on the horizon, including portability to new platforms and access to profiling information for intelligent optimization.

The CCA Forum [3] is addressing these problems by developing tools for component-based scientific computing in both high-performance parallel and distributed computing contexts. The CCA model and tools [4] provide language-neutral specification of common component interfaces, interoperability for software written in programming languages important to scientific computing, and dynamic composability, all with minimal runtime overhead. The current initiatives, which extend this preliminary work, are motivated by and will be validated through collaborations with a variety of computational science groups, including several SciDAC application teams.

5. Component interface issues

The greatest challenge in componentization is the definition of the components and their interfaces. The components need to be objects that are relevant for multiple implementation of beam dynamics simulations. They also need to have interfaces stripped bare of details that are specific to a particular implementation. Parallelization schemes complicate the issue; two different applications can agree that they both need a certain vector, but they differ in how they expect it to be distributed in a parallel computation.

A beam dynamics application taking on the order of a day on modern, massively parallel hardware would typically include propagating millions of particles through hundreds or thousands of accelerator (beamline) elements interspersed with hundreds of computationally expensive collective-effects calculations, all repeated a few thousand times. Since component calls inevitably have some overhead, calls that happen a few times *per collective effect* would be acceptable, but calls that happen even once *per particle* would be unacceptable.

The elements of beam dynamics that are common to essentially all simulations are the beam bunch and the beamline, as shown in figures 1 and 2. The beam bunch contains a dense array of particles. Since the coordinates of the particles are always relative to a reference particle, the bunch needs to carry the reference also. The beam bunch is the most fundamental component and, fortunately, the most straightforward to define. The beamline is also fundamental but, unfortunately, more problematic. Different beam dynamics implementations have very different ideas about how the elements comprising an accelerator should represented, split, and so forth. As such, it makes sense to have the entire beamline available as a component. One area of commonality betwen models is the representation of action of elements through Taylor maps. We have demonstrated interchanging CHEF and MaryLie beamline components at the map level, even though the beamline models themselves are very different in the two codes.

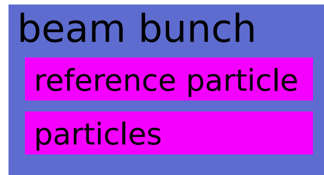

Figure 1. Beam bunch component with constituent reference particle and particles.

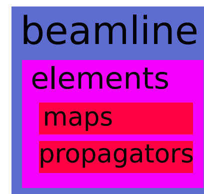

Figure 2. Beamline component with constituent elements, each possibly having a map and/or propagator.

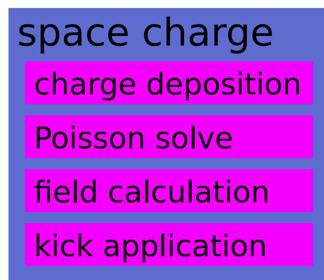

Figure 3. Space charge component with its constituent stages.

The bulk of our work to date on Synergia2 has been in the design and implementation of the space charge module, figure 3. Synergia2 can use the space charge modules from either IMPACT or Sphyraena. Granularity is an issue here. IMPACT and Sphyraena are interchangeable at the overall space charge level, but not at any finer level. The issue is the parallel decomposition. IMPACT uses a two-dimensional decomposition at the particle level, while Sphyraena uses a one-dimensional decomposition at the field level only.

The electron cloud component for Synergia2, Figure 4, is under development. The presence of several constituent stages that overlap with the constituent stages of the space charge component provides an opportunity for nontrivial code reuse. By carefully defining our interfaces, we are able to re-use aspects of the Sphyraena space charge module in the new electron cloud module.

6. Building and distributing componentized applications

While componentized applications represent an advance in capability and maintainability for scientific applications, they also suffer from the same problem facing all multielement software packages: build-time complexity. Figure 5 shows the non-trivial set of interdependent packages required to build Synergia2 and its constituent components. The traditional approach for distributing packages leaves the end user with a long list of dependencies to be built by hand. Our experience shows that expecting users to manually configure and build even a small set of dependent packages creates a substantial barrier to adoption.

We have created the tool Contractor [5] to automatically compile sets of dependent packages. Contractor is a *meta-build* tool; it compiles sets of dependent packages using their own native

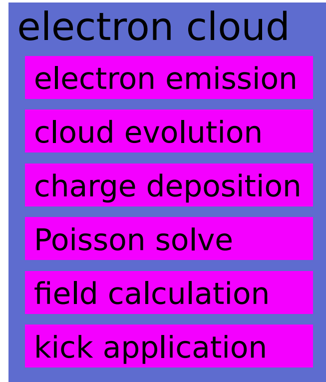

Figure 4. Electron cloud component with its constituent stages.

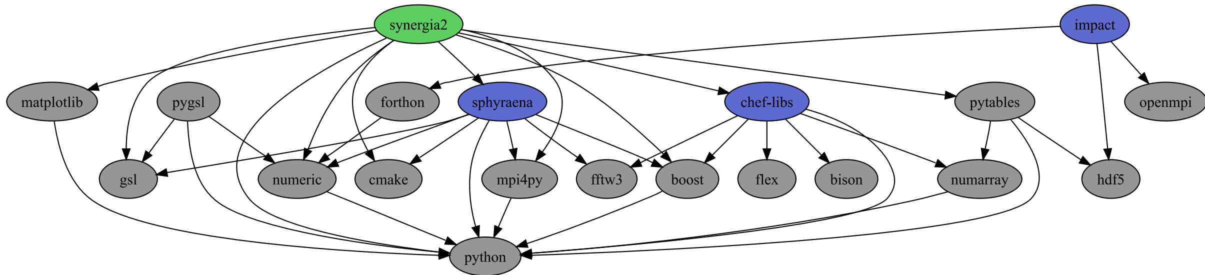

Figure 5. Packages required for building Synergia2, including component packages (blue) and infrastructure (gray).

build packages. In the simplest terms, make is a utility to invoke compilers such as cc and f90, on dependent sets of files. Contractor is a utility to invoke build systems such as GNU Autotools and CMake, on dependent sets of packages. Contractor is flexible enough to invoke arbitrary commands in a build process. It also has explicit support for GNU Autotools, CMake, Make, and the Python Setuptools.

The entire Synergia2 package and all of its dependencies can be compiled with a single Contractor command. The build can be interrupted and restarted. Contractor has a provision for user options, allowing the user to, for example, enable and disable optional packages or use system packages instead of compiling new versions. The state of the system can be queried by the user to see the status of options and the dependency tree itself. Figure 5 was generated from the Synergia2 Contractor description. In addition to being used for Synergia2, Contractor has been used as a meta-build system for CHEF, the CCA software, and other complex systems. We find it to be an invaluable tool for dealing with the build-time complexity arising in componentized applications during both development and distribution phases.

Acknowledgment
This work was supported in part by the U.S. Department of Energy under Contract DE-AC02-06CH11357.

References
[1] Ryne R, Dragt A, Decyk V, Habib S, Neri F, Mottershead C T, Samulyak R and Walstrom P MaryLie/IMPACT manual: A parallel beam dynamics code with space charge based on the MaryLie Lie algebraic beam transport code and the IMPACT parallel PIC code

[2] Amundson J, Spentzouris P, Qiang J and Ryne R 2006 *J. Comp. Phys.* **211** 229–248
[3] Common Component Architecture (CCA) Forum http://www.cca-forum.org
[4] Bernholdt D, *et al.* 2006 *Int. J. High-Perf. Computing Appl., ACTS Collection special issue* **20** 163–202
[5] Contractor Homepage http://home.fnal.gov/ amundson/contractor-www/index.html