The Common Component Architecture for Particle Accelerator Simulations

Douglas R. Dechow [*]
Tech-X Corporation, 5621 Arapahoe Ave., Suite A, Boulder, CO 80303
dechow@txcorp.com

Boyana Norris
Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439
norris@mcs.anl.gov

James Amundson
Fermi National Accelerator Laboratory, Batavia, IL 60510
amundson@fnal.gov

Abstract

Synergia2 is a beam dynamics modeling and simulation application for high-energy accelerators such as the Tevatron at Fermilab and the International Linear Collider, which is now under planning and development. Synergia2 is a hybrid, multilanguage software package comprised of two separate accelerator physics packages (Synergia and MaryLie/Impact) and one high-performance computer science package (PETSc). We describe our approach to producing a set of beam dynamics-specific software components based on the Common Component Architecture specification. Among other topics, we describe particular experiences with the following tasks: using Python steering to guide the creation of interfaces and to prototype components; working with legacy Fortran codes; and an example component-based, beam dynamics simulation.

Categories and Subject Descriptors D.2.13 [*Software Engineering*]: Reusable Software—domain engineering; J.2 [*Physical Sciences and Engineering*]: Physics

General Terms Design, Languages, Management, Standardization

Keywords Common Component Architecture, CCA, components, accelerator simulations, Synergia2, MaryLie/IMPACT

1. Introduction

As high-performance computing (HPC) platforms become more complex and software algorithms become more sophisticated, application scientists find it increasingly difficult to write physics software that takes full advantage of these developments. The (DOE) Office of Advanced Scientific Computing Research (OASCR) has recognized this problem and funded the SciDAC-2 (`http://scidac.org`) Center for Technology for Advanced Scientific Component Software (TASCS) [10] to build upon previous success in the development of the Common Component Architecture (CCA), a set of standards for component development that allows disparate components to be composed together to build a running application. This extensible component-based software architecture facilitates software interoperability between components developed by different teams across different institutions. The goal of TASCS (which funds the majority of work by the Common Component Architecture (CCA) Forum [7]) is to enable high-performance computing in DOE laboratories and elsewhere by introducing and supporting component-based software engineering practices and tools into scientific application development. The promise of CCA is that it will enable scientific applications to take fullest advantage of new supercomputing hardware by interfacing with specialized software components (for example, solvers for partial differential equations (PDEs)) that mathematics and computer science researchers have specifically tuned for maximum efficiency [6, 11, 16].

A perceived difficulty with introducing a component-based approach into scientific software is that computational science applications will need to be retrofitted to use these specialized software components, and this task can be too time-intensive for many researchers (both computer scientists and physical scientists). Lack of familiarity with component-based software engineering and lack of prominent examples within their topical areas is a barrier for investing the time to perform this retrofitting.

An example of a scientific software application where this effort has been undertaken is Synergia2 [4]. Synergia2 is a significant computational science application in the Office of High Energy Physics Accelerator SciDAC [12] Community Petascale Project for Accelerator Science and Simulation (COMPASS) [18]. Synergia2 is a beam dynamics modeling application for high energy accelerators such as the

[*] Correspondence should be directed to dechow@txcorp.com

HPC-GECO/CompFrame'07, October 21–22, 2007, Montréal, Québec, Canada.
Copyright © 2007 ACM 978-1-59593-867-1/07/0010…$5.00

Tevatron at Fermilab and the International Linear Collider (ILC), which is now under planning and development.

This article recounts the experiences of using the techniques and tools of the CCA framework to create a component based beam dynamics application based upon Synergia2. The development of a component-based variant of Synergia2 was accomplished via the following tasks:

1. Defining component interfaces using the Scientific Interface Definition Language (SIDL) and Babel [14] to address the multilanguage interoperability issues.
2. Creating components that adhere to the Common Component Architecture protocol.
3. Taking advantage of Python's lightweight characteristics for prototyping individual components.

2. Accelerator Modeling and Synergia2

Computational modeling and simulation of particle accelerators are enabling technologies for enhancing the full lifecycle of accelerators: analysis, design, optimization, and upgrading. Simulation tools for modeling the behavior of charged particle beams and their associated accelerators have traditionally been written as monolithic, problem-specific beam dynamics codes. The development of software that can support accelerator modeling activities is a complicated process. Additionally, government-funded scientific software has its own set of requirements that add complexity to the software development process. The Synergia2 application was created specifically to address some of these issues.

The Synergia2 application is a software tool developed at Fermilab for modeling the beam dynamics of high-energy particle accelerators. Synergia2 tracks the position and velocity of simulated particles as they move along the length of the accelerator. Synergia2 models (with analytic approximations) the various forces on the particles, such as the bending produced by dipole magnets, the focusing produced from quadrupole magnets, and the acceleration produced from cavities. An overview of the computational underpinnings of Synergia2 simulations is given in Section 2.1.

The algorithm for the magnetic force calculations used in Synergia2 simulations is from the *Chef* suite developed at Fermilab. Synergia2 also self-consistently models the repulsive forces between the particles (space charge forces) in three dimensions. The algorithm for these space charge calculations is from the IMPACT code [20], developed at Los Alamos National Laboratory and presently maintained at Lawrence Berkeley National Laboratory. Synergia2 is a very rare beam dynamics toolkit in that few accelerator simulation tools have self-consistent space charge, and an even smaller number of systems have three-dimensional algorithms.

Like many large, complex scientific software applications, the Synergia accelerator physics framework was developed in a non-traditional software engineering environment. A number of the algorithms at the core of Synergia, such as the calculation of magnetic forces on individual particles, are embarrassingly parallel. As a result, Synergia simulations are designed to execute on parallel architectures. Communication between the compute nodes is handled by the Message Passing Interface (MPI) [15]. Synergia is also a hybrid software application that is comprised of software libraries written in several programming languages. Adding to the complexity of the framework is the feature enabling Synergia simulations to be developed and executed in the high-level scripting language Python [19].

The software package diagram shown in Figure 1 gives some idea of the scale of the Synergia2 software development effort. In short, Synergia represents the current state of computational physics codes: parallel, multi-language, and complex.

2.1 Synergia2 Simulations

All modern charged particle accelerator simulation tools start with the definition of the accelerator lattice. The lattice is the physical layout of the machine as described by the organization of its functional units. The lattice is composed of a sequence of structures known as FODO cells [9]. A single FODO cell is composed of the following series of accelerator-specific optical devices:

- a **F**ocusing quadrupole magnet
- a drift space (denoted by the letter **O**)
- a **D**efocusing quadrupole magnet
- a second drift space (denoted by the letter **O**)

2.1.1 Describing Accelerator Lattices

The *lingua franca* of the accelerator simulation community is the MAD language. Originally developed to work with the Methodical Accelerator Design [13] program, the MAD language is a file format that is used to describe the physical layout of a charged particle accelerator. Among the important pieces of information that are contained in a MAD file are the following:

- the length and strength of individual beamline elements;
- the sequencing of lattice components;
- values assigned to variables for simple computation.

An example of the contents of a MAD file is shown below.

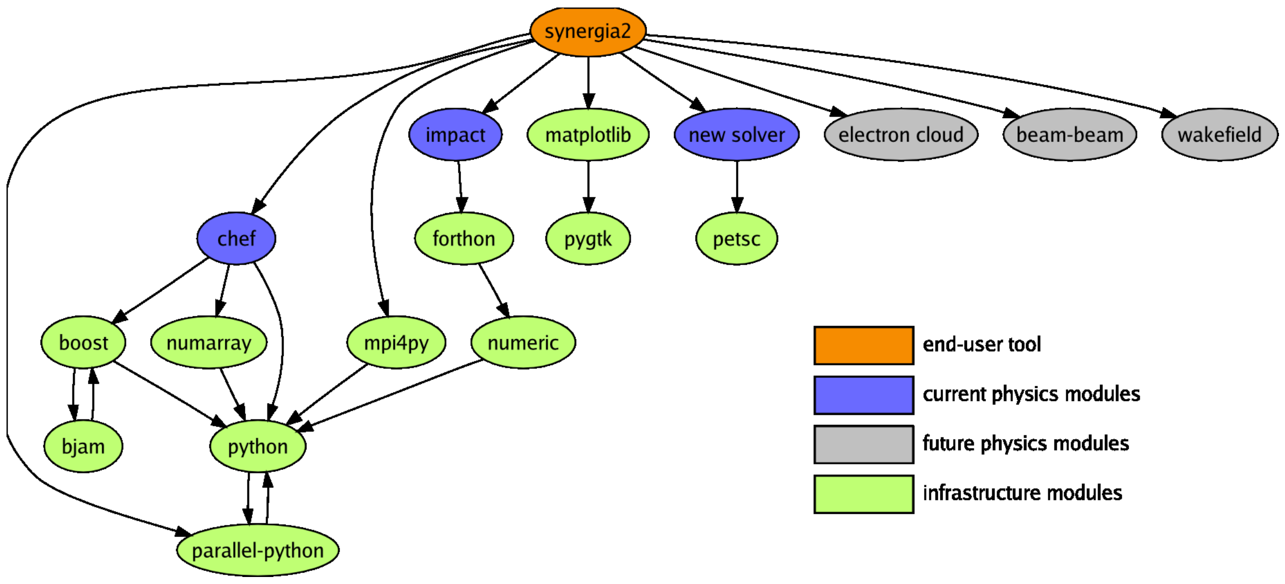

Figure 1. *Synergia2* packages

```
!scaling
scale=0.374749636

!drifts
drs: drift, l=7.44d-2
! "drift_short"
drl: drift, l=14.88d-2
! "drift_long" (equals 2*drs)

!quads
qd7: quadrupole, l=6.10d-2,
k1=-38.64d0/scale
qf7: quadrupole, l=6.10d-2,
k1= 38.64d0/scale
! line
channel: line=(drs, qd7, drl, qf7, drs)
```

This particular MAD file (*channel.mad*) is used for a range of Synergia2 simulations, and a simulation that uses it will be explored in greater detail in Section 4. All of the features of the MAD file language that were described above are demonstrated in this file. In particular, note that in the `quads` section that the variable `qd7` is defined to have a length attribute, `l`, and a strength attribute, `k1`. The value of `k1` is computed. Also, note that the defocusing quadrupole, `qd7`, is simply defined as the negative of the focusing quadrupole, `qf7`.

2.1.2 Split-Operator Algorithm

As was mentioned in Section 2, the Synergia2 application tracks the position and velocity of simulated particles as they move along the length of a charged particle accelerator. The computational algorithm that is at the heart of the Synergia2 application is based on a split-operator algorithm. The Synergia2 implementation of this technique has been detailed extensively elsewhere [4]. A brief overview is provided in the following paragraphs.

The key data structure in the Synergia2 application is a $7 \times n$ two-dimensional array where n is the number of particles used in the simulation. This data structure is used to model the phase-space of the beam bunch (i.e., particles). The phase-space representation is $[X, X', Y, Y', Z, Z', id]$, where X, Y, Z are the positional coordinates of a particle and X', Y', Z' are the momenta. The *id* is a unique, numeric identifier for each particle.

The split-operator technique relies on separating those operations that change the positional values of a particle from the operation of changing the momenta of a particle. Traditionally, beam dynamics simulations have used the concept of a transfer map to change a particle's position. A transfer map is a matrix representation of the magnetic properties of a beamline element (e.g., a focusing quadrupole). Using this representation, the application of a transfer map to a beam bunch becomes a matrix-matrix multiplication. To conform to the algorithm of the split-operator technique, the transfer map representing half the length of a beamline element is calculated and then applied to the beam bunch. Next, the momentum *"kick"*, or change, is calculated (making use of a Poisson Solver that is described in Section 3.2.4) for each particle in the beam bunch. Finally, the transfer map representing the second half of the beamline element is applied to the beam bunch.

2.2 Steered Simulation Interface

One relatively recent technique that has evolved to deal with the complexity of developing scientific software, in general, and scientific simulations, in particular, is the use of a high-level scripting languages such as Python [5]. In the case of scientific simulation tools, the end result is a

Table 1. The programming languages used by the Synergia2 scientific packages and libraries.

Packages	Python	C++	C	F90	F77
Synergia2-Simulations	X				
Synergia2-New Solvers		X			
Synergia2-IMPACT				X	
Synergia2-Chef		X			
MaryLie/ IMPACT				X	X
PETSc			X		

software system where the users are free to develop models in a programming environment that does not force low-level management of memory as in C++ or dealing with deprecated, but still prevalent, concepts such as Fortran's common blocks. Synergia2's steered simulation interface was developed to specifically address concerns of this type in the context of charged particle accelerator simulations.

That said, a consideration that was equally as important for moving Synergia2 to a steered interface was the desire to ameliorate the language interoperability issues that have been present in Synergia2 from its inception. As a mixed-language environment, Synergia2's core functionality depends upon the integration of two extant object-oriented accelerator modeling libraries (IMPACT written in Fortran 90 and Chef written in C++) with a high-performance numerical library, PETSc (and its newly developed FFTW interface), which was developed in the C programming language. All of these activities are to be coordinated and steered by a another, more human-friendly interface framework, written in Python.

Table 1 contains a matrix associating the primary Synergia2 project packages and libraries and their respective implementation languages.

To achieve these ends, the Synergia2 application depends heavily on the software glue characteristics of the Python scripting language as described in [17].

One outcome of the Synergia2 software development effort devoted to resolving language interoperability issues was to push the team in the direction of creating software components. These efforts have largely consisted of creating new, cleaner interfaces to our Fortran-based software. Once this was done, the Forthon Python-Fortran90 interoperability tool was used to generate Python bindings. This has also forced us to be cognizant of language interoperability issues when designing our newest software tools such as the *New Poisson Solver*. One technique that we have used to keep mindful of these issues is to create our unit tests in Python, and then use BOOST.Python to generate the appropriate language bindings.

In the future, the plan is for Synergia2 to migrate as much of its language interoperability issues as possible to language-independent component specifications, for which language interoperability is handled automatically by the Babel tool [14]. Unlike most language interoperability solutions, Babel specializes in high-performance support of Fortran and Fortran 77. In the Babel system, users define their types using the Scientific Interface Definition Language (SIDL). The Babel compiler then generates glue code for each language. The Babel runtime library provides basic facilities and infrastructure to keep the model consistent. At present, Babel supports Fortran, Fortran77, Python, Java, C, and C++ and so removes language interoperability concerns from the developers. The Babel tool is discussed again in Section 4.

Finally, an example showing how scientists can use the Python steering interface to develop Synergia2 simulations is shown in Section 4.

3. Introducing CCA components to Synergia2

We employed an incremental, top-down approach to our component design. Our initial high-level interfaces encapsulate large-grain functionality. This approach exploits the existing separation between the different underlying software packages and ensures that the component performance overhead is negligible.

3.1 Common Component Architecture

A comprehensive description of the CCA, including a discussion of how it differs from other component models, is available in [6]; here we present a brief overview of the CCA environment, focusing on the aspects most relevant to the subsequent discussion of our accelerator components design.

The specification of the Common Component Architecture [8] defines the rights, responsibilities, and relationships among the various elements of the component model. Briefly, the elements of the CCA model are as follows:

- *Components* are units of software functionality that can be composed together to form applications. Components encapsulate much of the complexity of the software inside a black box and expose only well-defined interfaces (ports).
- *Ports* are the interfaces through which components interact. CCA ports can be thought of as a class or an interface in object-oriented languages, or a collection of subroutines, or a module in a language such as Fortran 90. Components may provide ports, meaning that they implement the interface, or they may use ports, meaning that they invoke methods implemented by another component. Components that provide the same port(s) are

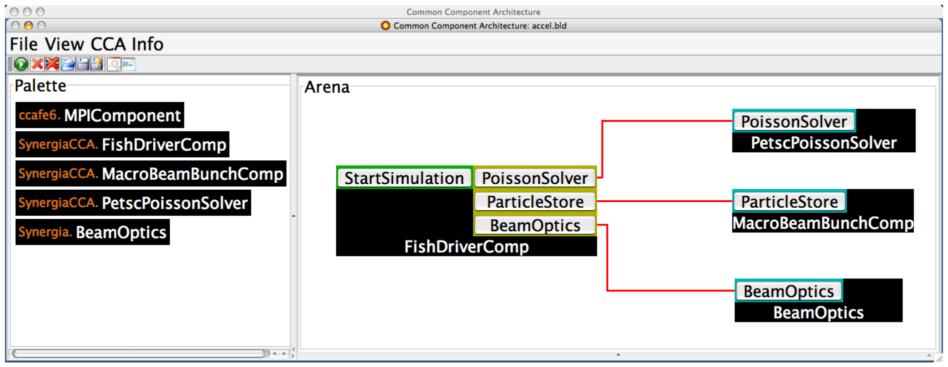

Figure 2. Component wiring diagram.

considered functionally equivalent and can thus be used interchangeably.

- *Frameworks* manage CCA components as they are assembled into applications and executed. The framework is responsible for instantiating components, destroying instances, and connecting *uses* and *provides* ports without exposing the components' implementation details. The framework also provides a small set of standard services that are available to all components.

3.2 Interfaces and Components for Particle Accelerator Simulations

In this section we discuss the ports we have defined and the corresponding components using or providing them. In this initial design, we are focusing on encapsulating large-grain functionality; thus, only three ports and a driver component are necessary for assembling a simulation. The component wiring diagram in Figure 2 shows an example application composed of components based on MaryLie/IMPACT, IMPACT, and the *New Poisson Solver* (which is based on PETSc and FFTW).

3.2.1 Python Simulation Driver

A *driver* component normally plays the role of a coordinator of the other components, in effect defining the top-level workflow of the application. The SIDL definition of the `FishDriverComp` component is shown below. The class body contains no methods because the use of the `implements-all` SIDL keywords signifies that all interface methods are implemented by this class without having to explicitly include them in the class declaration.

```
class FishDriverComp implements-all
          gov.cca.ports.GoPort,
          gov.cca.Component { }
```

This component (implemented in Python) provides the standard `gov.cca.ports.GoPort` interface, which is used to start the application's execution.

3.2.2 Particle Store

The `ParticleStore` port encapsulates tasks related to managing the representation of the particles in the beam.

```
interface ParticleStore extends gov.cca.Port
{
  void initialize( );
  void generate_particles(
           in double current,
           in int num_particles);
  void configure_beam(
           in double scale1,
           in double scale2,
           in double scale3,
           in array<double,2,row-major>
                 correlation_matrix);
  array<double,2,column-major>
           get_particles();
}
```

The components providing the `ParticleStore` port are as follows.

```
class BeamBunchComp implements-all
                advaccelsims.ParticleStore,
                gov.cca.Component { }

class MacroBeamBunchComp implements-all
                advaccelsims.ParticleStore,
                gov.cca.Component { }
```

The `BeamBunchComp` component is implemented in Fortran 90 and is based on IMPACT, while the `MacroBeamBunchComp` component is implemented in Python and is based on the *New Poisson Solver* package.

3.2.3 Beam Optics

The `BeamOpticsPort` port encapsulates tasks related to the application of transfer maps to the particles in the beam. The `BeamOpticsPort` interface is an example of an early interface design. It is essentially a one-to-one mapping of the ML/I routines that are implemented by the component.

```
interface BeamOpticsPort extends gov.cca.Port
{
void initialize();
void fquad3(in double l,
     in double gb0,
     inout array<double,1,column-major> h,
     inout array<double,2,column-major> mh);

void dquad3(in double l,
     in double gb0,
     inout array<double,1,column-major> h,
     inout array<double,2,column-major> mh);

void drift3(in double l,
     inout array<double,1,column-major> h,
     inout array<double,2,column-major> mh);
}
```

The `BeamOptics`, a Fortran90-based component, implements the `BeamOpticsPort` port.

In the future, we will be building upon our current experience by developing a more general interface that can support a wide-range of transfer map implementations (i.e., not just those implementations available in Synergia2 and ML/I). One possible incarnation of this interface is described by the prototype interface that is discussed in the following paragraphs.

The `Mapper` port defines a simple interface for applying transfer maps for changing a particle's position (as described in Section 2.1.2). The `apply` interface method will be used to change a particles positional values in the beam bunch abstraction. Components providing the `Mapper` port will also have a `ParticleStore` *uses* port to obtain the particle beam information.

```
interface Mapper extends gov.cca.Port
{
   void apply();
}
```

We have implemented the following two components providing the `BeamOpticsPort`:

```
class FastMappingComp implements-all
                       Synergia.BeamOpticsPort,
                       gov.cca.Component {}

class MLIMapperComp implements-all
                       Synergia.BeamOpticsPort,
                       gov.cca.Component {}
```

The `FastMappingComponent` component is based on the Lie algebraic mapping routines in the Chef single-particle, beam optics library. The `MLIMapperComp` component will be an evolution of the `BeamOptics` component and is based on an underlying set of Fortran transfer map subroutines present in MaryLie/IMPACT.

3.2.4 Poisson Solver

A super-fast and scalable Poisson solver is essential for beam dynamics simulations. This includes modeling high-intensity/high-brightness beams in linear accelerators and rings, modeling electron cloud effects in rings, and modeling beam-beam effects in colliders. To self-consistently model those effects, one must solve the Poisson equation at every time step in the beam dynamics codes.

The Synergia2 framework, which uses a Particle-In-Cell (PIC) method to solve the Poisson-Vlasov equation in three dimensions, currently implements two parallel linear solvers. Both solvers implement open, closed, and periodic boundary conditions. The first solver makes use of domain decomposition and a Fast Fourier Transform (FFT) provided by the IMPACT code [20]. In this context, domain decomposition is a process of assigning the particles to two-dimensional processor subdomains that represent a spatial region. A particle manager provides load balancing capabilities in this case. However, this FFT-based Poisson solver does not scale well on large numbers of processors because of global communication. Furthermore, the computational cost of the FFT-based solver normally scales as $N \log N$.

The second solver makes use of particle decomposition and is implemented via the PETSc library (in C). Additionally, the actual solve is performed via FFTW-based routines that have been integrated into the PETSc library. This *New Poisson Solver* has been made available as a CCA-compliant component.

As our work progresses, all of the current and future Synergia2 solvers will be available as software components. The ability to easily switch between solver implementations will allow the use of different techniques and new efficient algorithms.

The Poisson solver port for accelerator simulations is defined as follows.

```
interface PoissonSolver extends gov.cca.Port
{
  void solve( );

  void apply_space_charge_kick(
           in array<int,1> gridddim,
           in array<double,1> size,
           in array<double,1> offset,
           inout ParticleStore particles,
           in double tau);
}
```

Currently, the `apply_space_charge_kick` method is used to encapsulate a method of the same name and argument signature in the *New Poisson Solver*. The computational result of the method is a change in the momenta of the simulations particles.

The second method that is present in the interface, `solve`, is present as a placeholder for a planned, more general (i.e., non-beam dynamics specific) interface to what is hoped will become a highly performant solver.

We have implemented the following component wrapper for the *New Poisson Solver* (in Python) providing the `PoissonSolver` port:

```
class PetscPoissonSolver implements-all
            advaccelsims.PoissonSolver,
            gov.cca.Component { }
```

3.3 Fortran Experiences

In componentizing Fortran portions of the code (based on MaryLie/IMPACT), we encountered a number of challenges and identified some approaches that can be generally useful when creating SIDL-based component wrappers for any Fortran application. The following steps outline an overall design strategy for creating Fortran-based components.

- Create Fortran modules containing interface definitions that will be exposed through component interfaces. This is an opportunity to create better or different interfaces than offered by the legacy software.
- Define a new port for accessing the computations in the Fortran application. The SIDL port definition should match the Fortran module interfaces defined in the previous step.
- Define a component providing the port and use Babel to generate Fortran implementation skeletons, which will contain the calls to the original Fortran library.

A number of Fortran features commonly encountered in legacy codes present challenges in designing and implementing SIDL-based component wrappers. We enumerate some of the language features and software characteristics that make wrapping non-trivial and present our current or planned approach in providing the adaptor code between the SIDL-based component and the original Fortran library.

Non-Modular Software Creating a port (component interface) to encapsulate some functionality that is not already modular in the existing code can be challenging. In some cases simply mirroring existing library interfaces is a reasonable approach. In many cases involving older codes, however, the existing interfaces may not be ideal for the new component design, or there may not even be well defined interfaces, especially in Fortran 77 codes. In the later case we defined a new Fortran 90 module wrapping desired portions of the original library interfaces and corresponding component implementations.

Global Data By definition, components can only exchange information through well-defined interfaces. By contrast, many monolithic single-language applications rely on some global data representation for accessing the application's state. If new components introduce interface boundaries within a code that relies on global data, new interfaces must be created to enable all required information to be passed between these components. In the case of the accelerator software discussed in this paper, the new components were introduced at a granularity where global data is not used to exchange information. As the design is refined, however, we will have to encapsulate common block data and expose it through new interfaces.

Derived Types Derived types are frequently employed in Fortran 90 (and 95, 2003) modules to encapsulate module-specific state. When information encapsulated in derived types must be exchanged between components, SIDL mappings to these types must be defined as interfaces or classes, so that they can be used as argument types in port methods. In the current port definitions described in Section 3, all arguments are using built-in Babel types, including arrays, but future refactoring will most likely involve interfaces with SIDL-wrapped derived types.

A surprisingly difficult obstacle was presented by certain derived types, in the case when a variable of such a type must be included in the state of a component wrapping an existing library, such as MaryLie/IMPACT. Babel provides a mechanism for storing internal component state within a `sequence` derived type. For a `sequence` derived type, the order of the components specifies a storage sequence for objects declared with this derived type (compilers do not guarantee a particular order for non-sequence types). Sequence derived types are required by language interoperability tools such as Babel. The main difficulty in dealing with this restriction occurs when a non-sequence type in a legacy code (which cannot be modified) must be used for storing the state of a SIDL-based component wrapper. The only solution we found was to create an exact copy of the derived type definition, making it a `sequence` type, and then provide subroutines to copy to and from the original type. In the case of MaryLie/IMPACT, the type happened to encapsulate a relatively small amount of data and the copy routines do not impose a significant performance penalty. If the derived type happens to contain large arrays, this solution would not be feasible from a performance standpoint.

Arrays Babel provides two different data structures for managing arrays: SIDL arrays and r-arrays. A SIDL array is a regular SIDL type providing a generalization of the built-in array types available in different languages. Operations on SIDL arrays are performed using an API rather than built-in array functionality. It is possible, however, to operate on a native array type encapsulated by a SIDL array type in some of the languages, e.g., Fortran. R-arrays (or raw arrays) provide low-level access to numeric arrays in a subset of the Babel languages: C, C++, and Fortran.

We decided to use SIDL arrays for two reasons: (1) since some of the simulation components are implemented in Python, the interfaces cannot include r-arrays, and (2) in Fortran 9X, fewer changes to existing code are required when SIDL arrays are used since the native Fortran array can be obtained by using an array pointer and then used normally. By contrast, using r-arrays requires pervasive code changes since the array indexing starts at 0, while most Fortran codes assume 1[1]. Another array-related interface decision is whether each array should be column- or row-major. We selected column-major ordering for interfaces whose implementations will be predominantly in Fortran (e.g., the methods in the `ParticleStore` port), and row-major for those whose implementations are in likely to be in other languages (e.g., the `PoissonSolver` port) in order to eliminate or minimize array copying in the component wrappers.

3.4 Python Experiences

As was described in Section 2.2, the Synergia2 project has invested a significant amount of time and effort in developing Python-based tools. As a result of these experiences, it was only natural to explore the use of Python-based CCA components.

3.4.1 Leveraging Other Python Tools

A significant benefit to prototyping components using Python is that it has allowed us to maintain our investment in the Python tools that we already depend upon. For example, even though it is our current intention to replace our IMPACT-based beam bunch representation with a newer one that was developed simultaneously with the *New Poisson Solver*. Because the original IMPACT-based bunch had existing Python bindings (developed with the Forthon tool), we were able to use the older beam bunch until the newer beam bunch was mature enough to be componentized itself.

Additionally, the Python support in the CCA toolkit is flexible enough for use to continue to use all of our existing Python modules from a Python driver component just by importing them. This allowed us to develop an iterative programming style where we could mix and match component-based and non-component-based Python modules. This flexibility has given us the ability to test out our intuition concerning which pieces of our simulation codebase are appropriate component candidates, and which of our Python tools we will continue to use as is. An example of this is demonstrated in section 4.1 where the use of our current diagnostics module is discussed.

4. Example Simulation

One of the primary areas of accelerator physics that Synergia2 is concerned with is that of collective effects, sometimes also called collective instabilities [9]. Collective effects are oscillatory perturbations of the beam distribution which can cause the beam to "fall out of the pipe." For the Synergia2 project, we have adopted an operational interest in collective effects. We are interested in those collective effects that can be modeled by applying a momentum kick to the particles. In particular, Synergia2 is concerned with the space-charge collective effect. The space-charge collective effect is an outcome of intra-particle electromagnetic effects.

In Section 2.1.1 we described the contents of a MAD file. In this section, the same MAD file will be used as the basis for a Synergia2 simulation known as `openchannel`. Here, we will describe the primary component-level activities of the simulation (in the form of code snippets) as they relate to the split-operator technique described in Section 2.1.2.

At the component level, the simulation is executed by running the Ccaffeine framework script [2,3] shown below:

```
 1  #!ccaffeine bootstrap file.
 2  #———— don't change anything ABOVE this line.——
 3  path set /cca/techx/aas—cca/components/lib
 4  path append /cca/techx/mli/cca/components/lib
 5
 6  repository get—global
    SynergiaCCA.FishDriverComp
 7  instantiate SynergiaCCA.FishDriverComp
    SynergiaCCAFishDriverComp
 8
 9  repository get—global
    SynergiaCCA.MacroBeamBunchComp
10  instantiate SynergiaCCA.MacroBeamBunchComp
    SynergiaCCAMacroBeamBunchComp
11
12  connect SynergiaCCAFishDriverComp ParticleStore
    SynergiaCCAMacroBeamBunchComp ParticleStore
13
14  repository get—global Synergia.BeamOptics
15  instantiate Synergia.BeamOptics
    SynergiaBeamOptics
16
17  connect  SynergiaCCAFishDriverComp BeamOpticsPort
    SynergiaBeamOptics BeamOpticsPort
18
19  repository get—global SynergiaCCA.PetscPoissonSolver
20  instantiate SynergiaCCA.PetscPoissonSolver
    SynergiaCCAPetscPoissonSolver
21
22  connect  SynergiaCCAFishDriverComp PoissonSolver
    SynergiaCCAPetscPoissonSolver PoissonSolver
23
24  go SynergiaCCAFishDriverComp StartSimulation
25
26  quit
```

Ccaffeine [1] is a CCA-compliant framework for providing all of the needed runtime services for supporting a component infrastructure. In this script, the `path set` and `path append` commands tell the Ccaffeine framework where to locate the shared libraries for the user-created components.

The bulk of the script supports three component-level tasks:

1. loading libraries (lines 6, 9, 14, and 19), for example:

   ```
   repository get-global SynergiaCCA.MacroBeamBunchComp
   ```

2. instantiating components (lines 7, 10, 15, and 20), for example:

   ```
   instantiate SynergiaCCA.MacroBeamBunchComp
               SynergiaCCAMacroBeamBunchComp
   ```

3. and connecting *uses* ports to *provides* ports (lines 12, 17, and 22), for example:

   ```
   connect SynergiaCCAFishDriverComp ParticleStore
           SynergiaCCAMacroBeamBunchComp ParticleStore
   ```

The penultimate line of the script uses the `StartSimulation` command (an implementation of the standard `gov.cca-.ports.GoPort` port) to set the simulation in motion. The simulation code in the go method will then accomplish a

[1] While Fortran 9X allows lower bounds in each array dimension to start at any integer value, many loops in Fortran applications are hard-coded to begin iterating at 1.

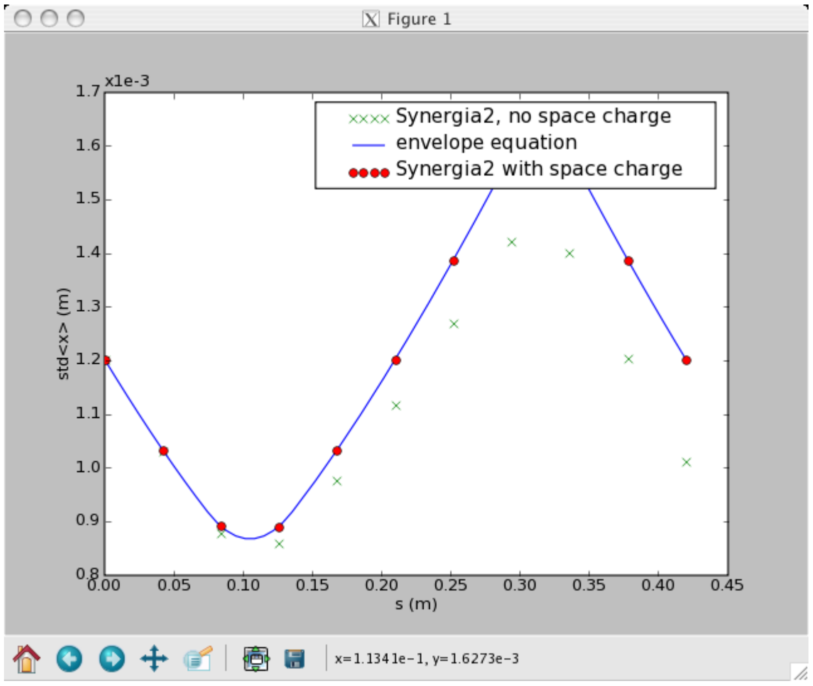

Figure 3. Envelope equation showing space charge effect

sequence of tasks. Those tasks are illustated below for a single, focusing quadrupole beamline element (using Python as pseudocode).

- beam bunch initialization:

  `beambunch = bunchport.initialize()`

- first-half transfer map application:

  `(h,mh) = beamport.fquad3(l/2, gb0, h, mh)`

- application of space charge kick:

  `solverport.apply_space_charge_kick(...,beambunch,..)`

- second-half transfer map application:

  `(h,mh) = beamport.fquad3(l/2, gb0, h, mh)`

4.1 Checking the Results

To confirm that our space charge routines are working as expected, we run a simple check based on an envelope equation. The envelope of a beam bunch is a mathematical description of the shape of the bunch. This is used as a pass/fail check, and an example of an expected result is shown in Figure 3. The envelope equation is an analytic approximation using the assumption that the beam's emittance does not change.

As was previously described in Section 3.4, one of the great things about prototyping component applications with Python is that existing Python tools can often be used within a Python driver component in a straightforward fashion. The following Python snippet illustrates the use of the Synergia2 diagnostics module. This is the code that produced Figure 3.

```
from diagnostics import Diagnostics
from loadfile import loadfile

d = Diagnostics()
d0 = Diagnostics("channel0current")

pylab.plot(d0.s,d0.std[:,diagnostics.x],'gx',
        label='Synergia2, no space charge')
pylab.xlabel('s (m)')
pylab.ylabel('std<x> (m)')

e = loadfile("env_matchchnl_0.5A.dat")
pylab.plot(e[:,0],e[:,1],
         label='envelope equation')

dold = diagnostics.Diagnostics(".")
pylab.plot(d.s,d.std[:,diagnostics.x],'ro',
         label='Synergia2 with space charge')
pylab.legend(loc=0)

pylab.show()
```

Figure 3 represents the envelope equation for a single FODO cell. Shown is a plot of the root mean squared of the observable portion of the x-coordinate of the beam bunch distribution versus the beam bunch's location in the accelerator lattice. In this example, the maximum beam compression effects of the focusing quadrupole can be seen at approximately 0.10 m. The defocusing quadrupole and its corresponding expansion effects are greatest at 0.32 m. The "no space charge" result is included as a comparison to demonstrate that the space charge routines are working.

5. Conclusion

We introduced our prototype design and implementation of a series of new beam dynamics-specific components for accelerator simulations. The components support the development of collective effects-based charged particle accelerator simulations: particle and bunch representation, particle tracking by transfer map application, and collective effects modeling in the form of space charge momentum kicks. The CCA framework provided the necessary language interoperability tools and component runtime infrastructure to deploy our prototype openchannel simulation. The flexibility and completeness of the CCA framework's support for the Python scripting language made the translation of an existing Synergia2 simulation a straightforward exercise. Future work plans include generalizing some of the port definitions and comparing the performance of component-based simulations to those using the original Synergia2 packages. Due to the coarse granularity of the new components, we don't expect the fixed component overhead to have a significant impact.

Acknowledgments

Portions of this work were supported by an SBIR Phase I research grant (grant number: DE-FG02-06ER84520) from the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy. This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357. The authors wish to thank the following people for their discussions and implementation work: Dr. Hong Zhang, Dr. Barry Smith, Dr. Dan Abell, Dr. Eric Stern, and Dr. Lois Curfman McInnes.

References

[1] B. Allan, R. Armstrong, S. Lefantzi, J. Ray, E. Walsh, and P. Wolfe. Ccaffeine – A CCA component framework for parallel computing. `www.cca-forum.org/ccafe`, 2005.

[2] B. A. Allan and R. C. Armstrong. CCA tutorial: Introduction to the Ccaffeine framework. `www.cca-forum.org/tutorials/archives/2002/tutorial-2002-06-24/tutorialM%odFramework.pdf`, 2002.

[3] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, and J. A. Kohl. The CCA core specification in a distributed memory SPMD framework. *Concurrency and Computation: Practice and Experience*, (14):1–23, 2002.

[4] J. Amundson, P. Spentzouris, J. Qiang, and R. Ryne. Synergia: An accelerator modeling tool with 3-D space charge. *Journal of Computational Physics*, 211:229–248, Jan. 2006.

[5] D. M. Beazley and P. S. Lomdahl. Building flexible large-scale scientific computing applications with scripting languages. 1997.

[6] D. E. Bernholdt, B. A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. W. Epperly, M. Govindaraju, D. S. Katz, J. A. Kohl, M. Krishnan, G. Kumfert, J. W. Larson, S. Lefantzi, M. J. Lewis, A. D. Malony, L. C. McInnes, J. Nieplocha, B. Norris, S. G. Parker, J. Ray, S. Shende, T. L. Windus, and S. Zhou. A component architecture for high-performance scientific computing. *Intl. J. High Perf. Comp. Appl.*, 20(2):163–202, 2006.

[7] CCA Forum homepage. `www.cca-forum.org`, 2007.

[8] CCA specification. `cca-forum.org/specification`, 2007.

[9] M. Conte and W. M. MacKay. *An Introduction to the Physics of Particle Accelerators*. World Scientific, Singapore, 1991.

[10] David Bernholdt (PI). Technology for Advanced Scientific Component Software (TASCS). SciDAC2 CET, `http://www.scidac.gov/compsci/TASCS.html`, 2006.

[11] L. C. M. et al. *Numerical Solution of Partial Differential Equations on Parallel Computers*, chapter Parallel PDE-Based Simulations Using the Common Component Architecture, pages 327–381. Number 51 in Lecture Notes in Computational Science and Engineering. Springer, 2006. Preprint ANL/MCS-P1179-0704.

[12] R. R. et al. SciDAC advances and applications in computational beam dynamics. *Journal of Physics Conference Series*, 16:210–214, Jan. 2005.

[13] H. Grote, F. C. Iselin, E. Keil, and J. Niederer. The MAD program. In *Particle Accelerator Conference, 1989. 'Accelerator Science and Technology'., Proceedings of the 1989 IEEE*. IEEE, 1989.

[14] Lawrence Livermore National Laboratory. Babel. `www.llnl.gov/CASC/components/babel.html`, 2007.

[15] MPI Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4):159–416, Fall-Winter 1994.

[16] B. Norris, S. Balay, S. Benson, L. Freitag, P. Hovland, L. McInnes, and B. Smith. Parallel components for PDEs and optimization: Some issues and experiences. *Parallel Computing*, 28(12):1811–1831, 2002.

[17] J. K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, 1998.

[18] Panagiotis Spentzouris (PI). Community Petascale Project for Accelerator Science and Simulation (COMPASS). FNAL DOCDB, CD-doc-2098, version 1, 2007.

[19] Python programming language – Official website. `python.org`, 2007.

[20] J. Qiang, R. D. Ryne, S. Habib, and V. Decyk. An Object-Oriented Parallel Particle-in-Cell Code for Beam Dynamics Simulation in Linear Accelerators. *Journal of Computational Physics*, 163:434–451, Sept. 2000.