artG4: A Generic Framework for Geant4 Simulations

Tasha Arvanitis[1] and Adam Lyon

Fermi National Accelerator Laboratory
MS 357
Batavia, IL 60510-0500 USA

Correspondance e-mail: lyon@fnal.gov

Abstract. A small experiment must devote its limited computing expertise to writing physics code directly applicable to the experiment. A software "framework" is essential for providing an infrastructure that makes writing the physics code easy. In this paper, we describe a framework for writing Geant4 based simulations called "artg4". This framework is a layer on top of the *art* framework.

1. Introduction

Small experiments, such as the Fermilab Muon g-2 experiment with about 125 collaborators, do not have the computing and software expertise found at large experiments such as CMS and ATLAS. Large experiments have the personnel to write their own software infrastructure that physicists use to code algorithms and analysis programs. Small experiments generally have a few software experts; not enough to create an entire software ecosystem like those from the LHC experiments. Instead, they rely on software frameworks from other sources. The *art* framework[1] developed by the Fermilab Scientific Computing Division, is such a framework used by several experiments, including Muon g-2, Mu2e, NOvA, and Darkside-50. *art* is also the underlying framework for LarSoft used by LBNE, MicroBoone, and other liquid argon neutrino experiments.

[1] Current address: Harvey Mudd College, Claremont, CA 91711 USA

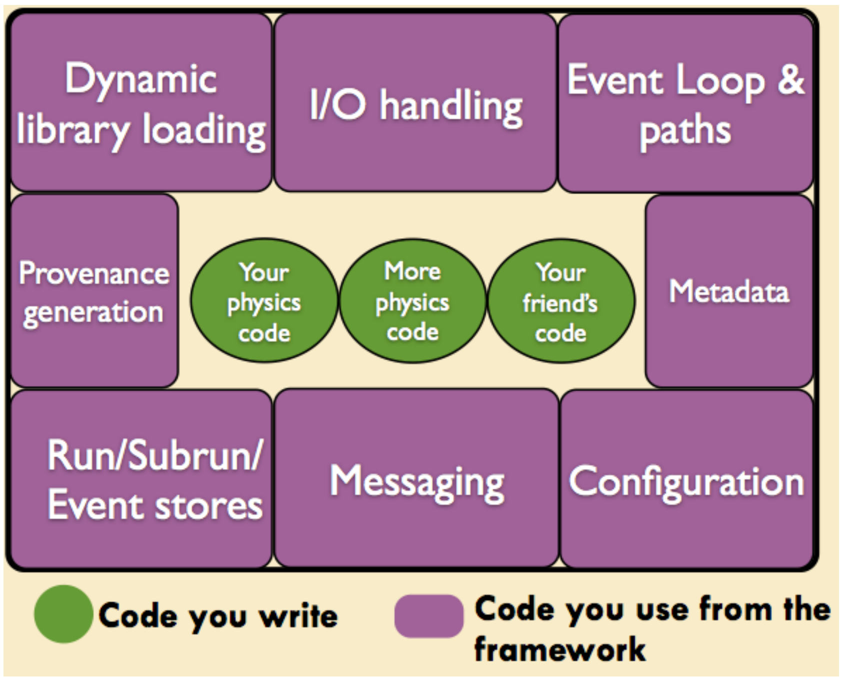


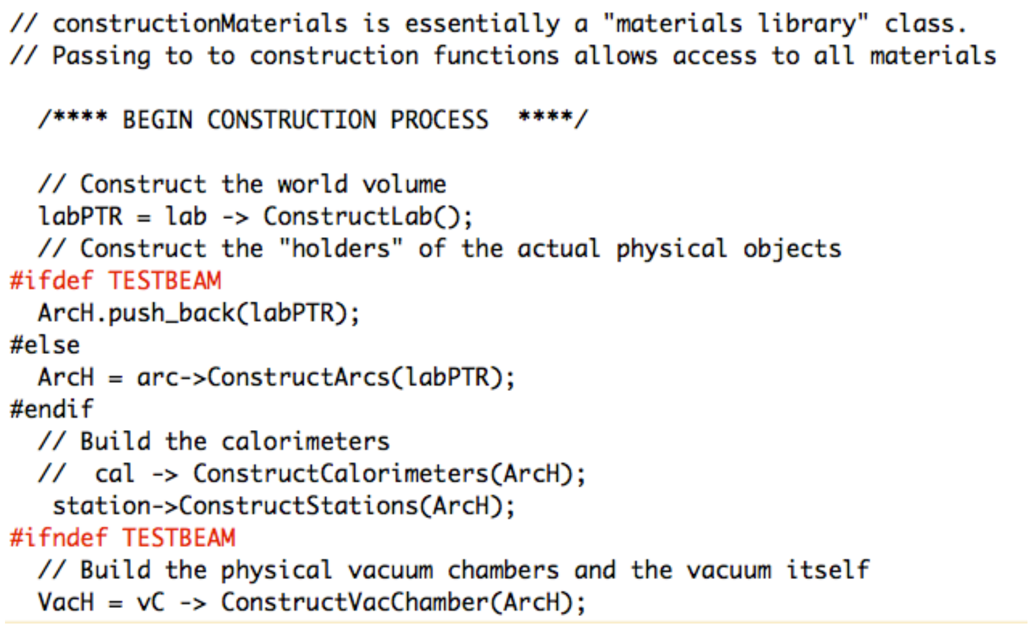
Figure 1. The functions of a framework are displayed, showing that code you write (as a physicist) lives within an infrastructure that you exploit.

Frameworks provide more than software infrastructure. They provide a mechanism for collaboration and doing what physicists really want to do - produce physics results. Figure 1 displays some of the functions of a framework system. Physics code lives within, exploiting the common infrastructure. For *art*, such physics code lives in dynamically loaded modules. Many such modules can be strung together and configured by the framework's configuration system, forming an analysis program designed to perform a physics task. Though modules may be written by different people, they have to conform to the rules of the framework, and so work together.

The framework provides the functionality that belong to the realm of computing experts, not physicists. For example, *art* is written by a group of C++ and framework experts at Fermilab. The low-level functions for i/o, dynamic loading, module dispatch, messaging and logging, data storage, and meta-data generation (including event provenance) need to be reliable and optimized. Computing experts are better equipped to succeed in those areas than particle physicists. The *art* team has indeed succeeded in writing such functionality and using the latest tools with C++ 2011.

Users of such frameworks generally have two reactions. Some, typically those with more computing expertise, find such systems constraining. The infrastructure is hidden behind the scenes, they may have ideas that were not considered by the experts, they have to trust a system they didn't write, and they missed out on what they consider the fun of writing the complicated code to do the low-level functions. Others find a framework system such as *art* liberating. They can concentrate on physics code instead of the infrastructure. The code they write is generally less complicated than the infrastructure code (they are using a complicated system, not writing it). They do not have to maintain the infrastructure code. With a framework, they can use code from others and share their code easily. And finally they get the services the infrastructure provides for free (such as data management).

For simulations based on a system like Geant4, a framework like *art* is especially useful for the reasons listed above. But Geant4 needs to be integrated with the framework, and the integration can provide functionality for ease of use on its own. The topic of this article is such an integration layer called *artg4* written by the Fermilab Scientific Computing Division for the Muon *g-2* experiment. Below we explore the motivation and details of *artg4*, how it integrated *Geant4* and *art*, and features that makes *Geant4* easier to use.



```
// constructionMaterials is essentially a "materials library" class.
// Passing to to construction functions allows access to all materials

  /**** BEGIN CONSTRUCTION PROCESS  ****/

  // Construct the world volume
  labPTR = lab -> ConstructLab();
  // Construct the "holders" of the actual physical objects
#ifdef TESTBEAM
  ArcH.push_back(labPTR);
#else
  ArcH = arc->ConstructArcs(labPTR);
#endif
  // Build the calorimeters
  //  cal -> ConstructCalorimeters(ArcH);
   station->ConstructStations(ArcH);
#ifndef TESTBEAM
  // Build the physical vacuum chambers and the vacuum itself
  VacH = vC -> ConstructVacChamber(ArcH);
```

Figure 2. Example of detector construction code with compiler flags

2. Motivation

artg4 was written in the context of the new Fermilab Muon *g-2* experiment. Measurement of the fundamental parameter *g-2* of the muon is a decades long industry. The latest result by the E821 experiment at Brookhaven National Laboratory from 1999-2001 is a 0.54 ppm measurement and is ~ 3 sigma away from the Standard Model prediction. As an intriguing hint of new physics, Fermilab is preparing to do the experiment again with quadruple the precision. If the difference between experiment and the theoretical prediction persists, it will be at the level of 5-7 sigma; an undeniable signal of physics beyond the Standard Model. Muon *g-2* is a high priority experiment for Fermilab and is currently scheduled to start taking data in 2017. The 50' diameter storage ring used in the E821 experiment has been moved to Fermilab and new detectors are currently under design and test.

Soon after Brookhaven completed the E821 experiment described above, several design studies were initiated to explore improvements to the muon storage ring for a follow-up experiment at BNL that never materialized. The result of these studies that remains is a Geant4 based simulation of the ring and detectors called *g2migtrace*[2]. *g2migtrace* is a very detailed and accurate simulation of the storage ring including beam injection and orbit dynamics as well as many ring and sensitive detector components. The Fermilab experiment adopted this software as its main tool for simulation studies.

While *g2migtrace* proved valuable for many simulation studies, the program is a monolithic code with a mostly hard-coded geometry and detector configuration. Interaction with the simulation is via the Geant4 messenger facility and command prompt. Given the monolithic nature, altering the code to try new ideas involves switches and many *if* statements.

g2migtrace includes a detailed simulation of the calorimeters used in the E821 experiment. Fermilab had plans to explore different calorimeter designs, and so a simulation of the reference E821 calorimeter was desired. *g2migtrace* has this simulation, but it is buried in the ring code. To create such a simulation, a version of *g2migtrace* with only the calorimeter was created by introducing compiler flags as shown in figure 2.

This “TESTBEAM” flag was scattered throughout the code to effectively remove all non-calorimeter simulation lines. While this idea worked in this instance, it essentially renders the code unmaintainable. How would one implement a different test beam configuration? How to add new detectors while keeping old ones for comparison? One imagines a proliferation of compiler switches causing a huge mess. Clearly, a framework with modules could rescue this situation.

3. Design of *artg4*

The idea behind using *art* and *artg4* was to use a framework that could modularize the *g2migtrace* simulation. We imagined having each detector component (e.g. the ring, calorimeters, trackers, beam position detectors, etc) in a module. Other aspects of the simulations, such as the particle guns and various Geant4 actions, would be modularized as well. Having a library of such modules, a simulation program could be stitched together with the framework configuration system. If one wanted to simulate the entire ring and detectors, all of those components would be configured in the simulation. If one wanted a test beam of one calorimeter, that too could be accomplished by writing a configuration file that invoked only the necessary modules to shoot a particle at one calorimeter. Because *g2migtrace* has a huge amount of valuable detailed and accurate code simulating these components, we embarked on a reorganization project instead of a re-write project.

4. Details of *art*

The *art* framework supplies all of the framework services that appear in figure 1. More details may be found in reference [1]. *art* uses a typical event model to describe physics data. Data objects with information about a particular event are accessible from an event store in memory. *art* modules process these data with algorithms and plot the results and/or can store them back into the event as new data objects. Physics code lives in these modules that *art* loads dynamically. There are three types: *producers, filters,* and *analyzers*. Producers and filters can access the event data and store new data into the event. In addition, filters return a Boolean value that can influence further processing of the event. Analyzers cannot write to the event and are meant for making diagnostic plots. Data are currently serialized by Root’s[3] i/o system.

Modules have the expected methods to handle different points in processing, including begin job, begin run, processing an event, end run, and end job. An important rule is that modules must be independent. That is they can only pass information via data in the event. This restriction ensures that modules are compatible with multithreaded execution.

A *service* is a globally accessible object, like a singleton, to which *art* enables access and is dynamically loaded. The object is completely generic and can do most anything, so long as it conforms to the restriction in the preceding paragraph. That is a module must not store information into a service that would be accessed by a module downstream. Any module can access any service. Example services are those serving Geometry information and constants, serving random numbers, and recording timing and memory usage information.

art uses a configuration language called “FHICL” (Fermilab Hierarchical Configuration Language) which is a structured text document that specifies which modules and services are loaded, their parameters, and other configuration information (e.g. module ordering).

5. Geant4 in *art*

Adapting Geant4 code into *art* leads to interesting challenges. Geant4 maintains its own event store and event loop. The first step in integrating Geant4 with *art* is to have an *art* module call parts of the *beamOn* method in *G4RunManager* in order to take control of the event loop. Geant4 must be configured with detectors constructed and actions specified. The naïve implementation would be to split detectors and actions into different modules, but then those modules would be communicating via

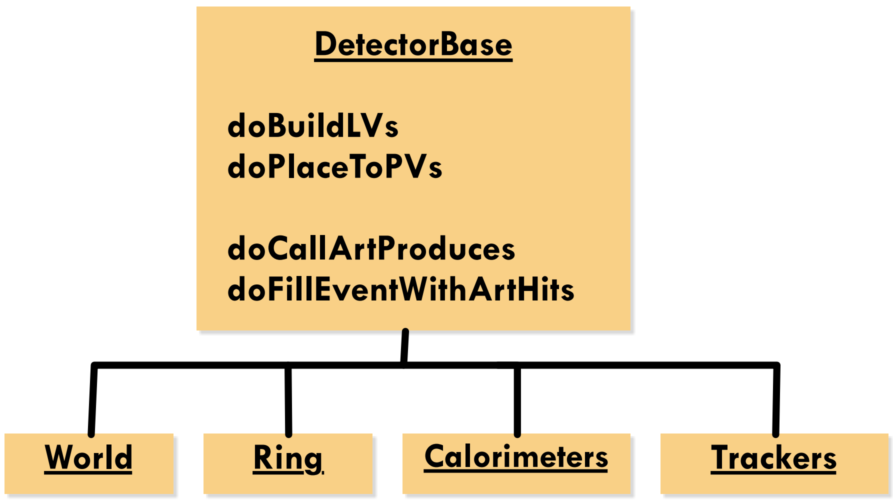


Figure 3. The structure of detectors.

the Geant4 event store instead of the *art* event store. Such communication violates the rules imposed by *art*. A better implementation involves having one *producer* module and dynamically loaded services to handle the different parts of the simulation. Because there is only one module dealing with Geant4 information, the *art* rule is not violated. This one-producer many-services structure will be used in *artg4*.

There are two basic pieces to a Geant4 simulation: Detectors and Actions. *g2migtrace* constructed detectors with code as opposed to GDML files. We decided to maintain that functionality. Constructing a detector involves defining its shape, creating a logical volume by adding material information, and finally creating a physical volume from the logical by adding placement information. Actions are hooks into Geant4 processes. For example, one can have user code called whenever a new event is started, whenever a track is created, and when the first particles of the simulation are generated.

For detector construction, the user writes objects as services that inherit from our *DetectorBase* class, as shown in figure 3. *DetectorBase* registers the detector object with a cataloging service so that the system need not know about all of the detectors *a priori*. One must override the member functions *doBuildLVs* and *doPlaceToPVs* to ensure that the steps of creating logical and physical volumes are followed. Each detector has parameters of *category* and *motherCategory* to define the detector hierarchy (the "world" detector has a blank *motherCategory*). If a detector has a sensitive element and produces data, those data must be converted from Geant4 objects into objects compatible with storage in the *art* event (e.g. without internal pointers). That conversion happens in the *doFillEventWithArtHits* method. The *doCallArtProducts* is a special one to alert *art* to expect data of the specified type.

Actions are also specified as services that inherit from a particular *ActionBase* class (for example *EventActionBase*). Member functions to override depend on the *ActionBase* chosen.

With this scheme, nearly all of the complicated sensitive detector code remains unchanged. These base classes and other infrastructure code make up *artg4*.

6. Adapting *g2migtrace* to *art* with *artg4*

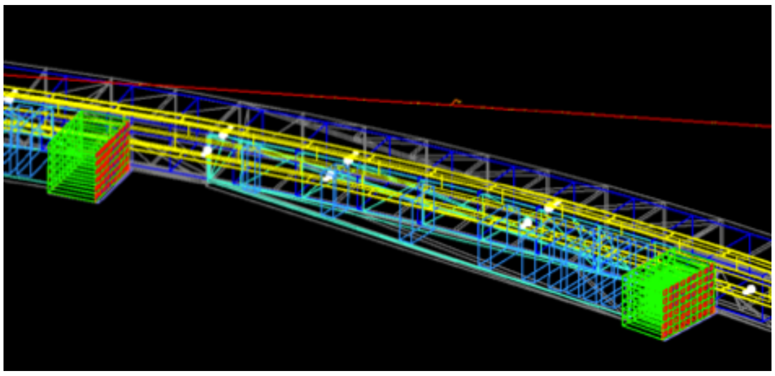


Figure 4. A section of the Muon *g-2* storage ring with calorimeter stations.

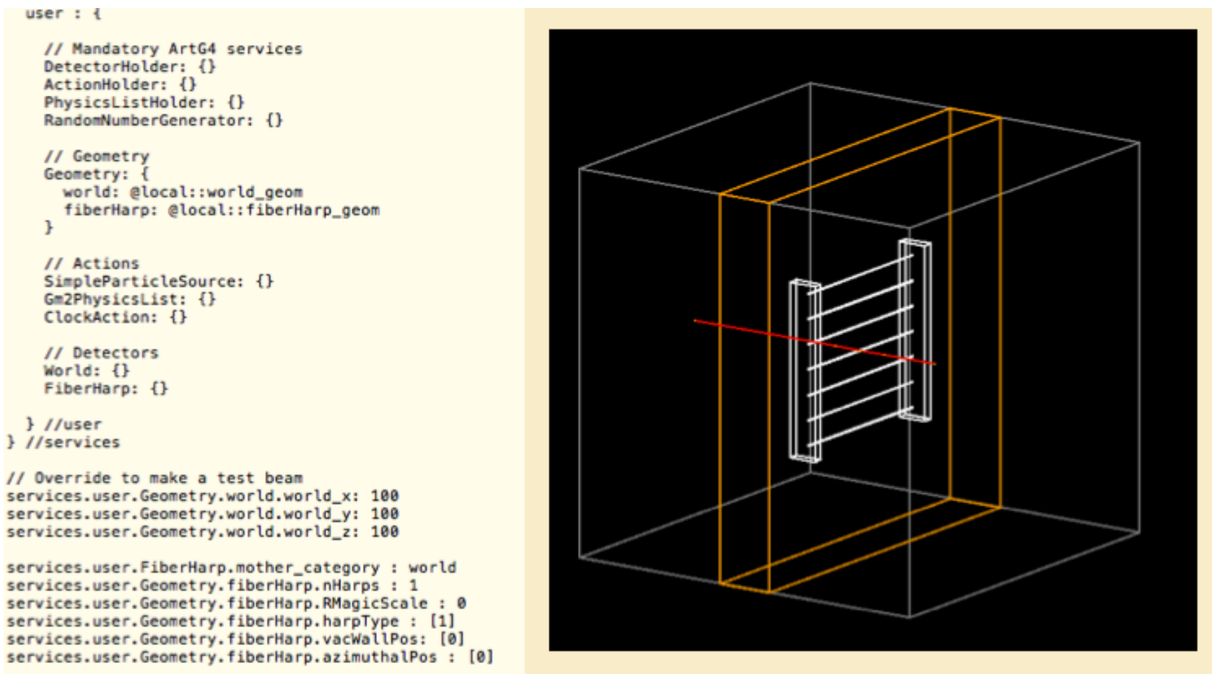
The main requirements for adapting *g2migtrace* to *art* were to have a modular simulation that is defined by the configuration file. These demands were easily achieved by *artg4*. As a practice, several members of the computing group converted the Geant4 N02 and N04 examples to the *artg4* framework. These studies were a great success and so a group of about five people tackled converting the *g2migtrace* code. It took us about 2 months to compete this task and the results were very successful. Figure 4 shows a section of the storage ring along with calorimeters constructed with the simulation.

One of the improvements we wanted to realize was the ability to constructed test beam or other simulations without massive changes to the code (e.g. no compiler flags). We were in fact able to accomplish this goal quite easily with *art* and *artg4* and the configuration language. Figure 5 shows a snip-it of the configuration file for a "Fiber Harp" (beam position monitor) test beam along with a visualization. The detector elements not needed for this simulation are not configured and so are not loaded by *art*. The *FiberHarp* service is the same one as is used by the ring simulation, but several parameters are overridden to create the test beam. The code in the *FiberHarp* service remains unchanged.

7. Conclusions

We were able to realize our goal of having an extremely flexible simulation system based on *art*. The flexibility and ease of use that the *art* framework provides enables many non-computing experts to participate in simulation writing and studies. Indeed we now have a group of about fifteen people actively using *artg4* in order to do simulations and studies. Sharing code and collaborating is very simple and is proving to be extremely fruitful for the experiment.

We wish to acknowledge the support from the Fermilab *art* developers and the Muon *g*-2 collaboration. The Fermi National Accelerator Laboratory is operated by Fermi Research Alliance, LLC under Contract No. De-AC02-07CH11359 with the United States Department of Energy. This contribution is FERMILAB-CONF-13-480-CD.



```
  user : {

    // Mandatory ArtG4 services
    DetectorHolder: {}
    ActionHolder: {}
    PhysicsListHolder: {}
    RandomNumberGenerator: {}

    // Geometry
    Geometry: {
      world: @local::world_geom
      fiberHarp: @local::fiberHarp_geom
    }

    // Actions
    SimpleParticleSource: {}
    Gm2PhysicsList: {}
    ClockAction: {}

    // Detectors
    World: {}
    FiberHarp: {}

  } //user
} //services

// Override to make a test beam
services.user.Geometry.world.world_x: 100
services.user.Geometry.world.world_y: 100
services.user.Geometry.world.world_z: 100

services.user.FiberHarp.mother_category : world
services.user.Geometry.fiberHarp.nHarps : 1
services.user.Geometry.fiberHarp.RMagicScale : 0
services.user.Geometry.fiberHarp.harpType : [1]
services.user.Geometry.fiberHarp.vacWallPos: [0]
services.user.Geometry.fiberHarp.azimuthalPos : [0]
```

Figure 5. A fiber-harp test beam with only changes to the configuration file and not the code itself.

References

[1] Green, C. and Kowalkowski, J. and Paterno, M. and Fischler, M. and Garren, L. and others 2012 *J. Phys. Conf. Ser.* ***396*** *022020*

[2] *g2migtrace* (unpublished) was written by Kevin Lynch (currently at CUNY York College, Jamaica, NY, USA) and Zach Hartwig (currently at MIT, Boston, MA, USA)

[3] Rene Brun and Fons Radmakers 1997, Inst. & Meth. in Phys. Res. A **389** 81-86