Fermi Offline Software:  
The Pros and Cons of Reusing Free Software

H Kelly
SLAC National Accelerator Laboratory Menlo Park, CA
E-mail: heather@slac.stanford.edu

Abstract. The Fermi Gamma-ray Observatory, including the Large Area Telescope (LAT), was launched June 11, 2008. We are a relatively small collaboration, with a maximum of 25 software developers in our heyday. Within the LAT collaboration we support Red Hat Linux, Windows, and are moving towards Mac OS as well for offline simulation, reconstruction and analysis tools. Early on it was decided to use one software system to run our simulations as well as ultimately handle the event processing for real data. We leveraged many existing HEP external libraries (Geant4, Gaudi Framework, ROOT, CLHEP, CMT) to ease the burden on our developers. This strategy of re-using existing software helped us pull together our system quickly and test during our beam tests and data challenges. Now, after launch, we are in a new phase of the project, where we must move forward to support modern operating systems and compilers to get us through the life of the mission. This means upgrading our external libraries as well, which are not under our direct control. Meanwhile, it is crucial to our production system that we carefully orchestrate all upgrades to insure stability. An additional hurdle is that our number of active developers has dwindled dramatically. Many of those left are Windows developers reliant on the Visual Studio development environment, while our user base and production system depend on our Linux distributions. There have been a number of lessons learned, with undoubtedly more to come.

1. Introduction
Work on the Fermi space telescope began long before the proposal phase in 1999. Early on, C++ was adopted as the preferred programming language for detailed Monte Carlo simulations. At that time, Geant4 was not yet released. Members of our team developed Gismo, an object oriented Monte Carlo [1]. Using Gismo’s geometry library, we built a description of the likely detector elements where alternative designs were rapidly prototyped and tested via simulations. Initial versions of our reconstruction software were also written during this period.

All simulation efforts were strongly windows-centric. Most, if not all, developers were using Visual Studio. Visual Source Safe was our code repository. An ASCII ntuple was the sole output of the simulations.
2. Post-Proposal Phase

After the Fermi space telescope became an accepted project, our paradigm quickly changed. There would be one code system to be used for simulations, test data analysis, and flight operations. Our development team swelled to a maximum of twenty-five active contributors, distributed across nine time zones. The software was ported to Linux and we moved to CVS for our code repository. Work began to re-write the reconstruction algorithms. With new developers and rapidly changing code, additional tools and infrastructure were necessary to make the code easily available and facilitate builds. A release manager was developed which automatically built the software in response to CVS tags. This freed developers who preferred to work with a particular OS/compiler to avoid testing on both Windows and Linux. Additionally, changes that caused build failures were immediately identified. The resulting binary releases were also available to the users who preferred not to bother building the code themselves.

A primary piece of the work to create a quality software system was the decision to utilize a number of external libraries. By leveraging those components we could reuse from other projects, attention could be focused on those portions of the software unique to Fermi. Fortunately, a number of high quality libraries were quickly becoming freely available such as: Geant4 [2], ROOT [3], and Gaudi [4].

3. Geant4

We were keeping our eye on the Geant4 development since the time of its inception. By 2000, it was clear that we needed to abandon our hand-crafted Monte Carlo toolkit and re-join the legions of Geant users. Migration was not simple, due to our hard-coded geometry implementation using the Gismo geometry primitives. However, it was at this time that our geometry description migrated to XML, parsed via Xerces, and at runtime geometry descriptions were generated.

Some of the advantages of Geant4 include its active development community, vetted physics simulation results, and commercial quality online documentation. One minor drawback has been the need to keep the version of CLHEP [5] in lock step with Geant4’s production version. We have another external library, Gaudi that also uses CLHEP as an external. There have been some conflicts between the requested CLHEP versions over the years. Recently, Geant4 chose to ingest those portions of CLHEP necessary for its suite of libraries. This coupled with the move to support CMake has made building Geant4 much easier, especially on Windows. Previously, we were using Cygwin to handle our Geant4 distributions which was much more cumbersome.

4. Gaudi Framework

A major issue associated with our existing simulation code was due to poor data handling. While we attempted to utilize C++ object oriented programming practices; data hiding was not necessarily congruent with our needs to ferry the data through a simulated detector response, reconstruction analysis and finally output. That coupled with inconsistent levels of object-oriented programming proficiency within our team, led to some very poorly designed code which was almost impossible to maintain. The Gaudi Framework philosophy suggested a different paradigm: separation of data from the algorithms. This was more appropriate for the typical needs of a Physics experiment.

Gaudi provided a number of services right out of the box, including: Transient Data Store (TDS), Persistency Service, JobOptions service, standard event loop, messaging, and logging. The TDS is fundamentally a mechanism to share memory. Data objects are registered with the TDS and are then available to any other algorithm, service, or component in the system. The persistency service provides I/O facilities for permanent storage of data. JobOptions refers to input parameters that are read in at run-time, which may be stored as a text file or python script. Since Geant4 also provides its
own event loop, we had to find a way to allow Gaudi to control the event loop by creating a special class that derives from Geant4’s G4RuntimeManager.

Migrating to the Gaudi framework required a major reorganization of our code, which was badly in need of reorganization anyhow. Figure 1 shows a high level diagram of the simulation/reconstruction system after the software re-write. Gaudi provided a simple interface we could adopt for our algorithms, services, etc.

The primary disadvantage the Gaudi framework has been the availability of updated detailed documentation. Gaudi also uses a large of number external libraries by default; twenty two are required for the v21r7 release. We chose to work around this by making minor modifications to the Gaudi source code. In this way we were able to limit ourselves to eight externals, where three of them we were already using as part of our own set of external libraries.

Building Gaudi is typically done via the Code Management Tool (CMT), which we adopted as well for building our software. The biggest hurdle is assembling the necessary externals for Gaudi for both Windows and Linux. In general, it is the windows builds that cause the most grief. Gaudi is also looking at CMake as a possible build tool and we would gladly take advantage of that development. In the future though, Gaudi may drop Windows support entirely. This will not affect Fermi as we are nearing the end of the mission and will likely freeze on a current production version of Gaudi which still provides Windows support.

Figure 1. Diagram of the Fermi offline simulation/reconstruction software after reorganization
5. ROOT Analysis Framework

Fermi adopted ROOT as its output format for use within the LAT team. Our ASCII ntuple was transformed into a ROOT ntuple. In addition, we were able to take advantage of ROOT’s object oriented I/O and store our full Monte Carlo, detector response, and reconstruction quantities within separate ROOT files. The Fermi data is mandated by the HEASARC to be provided to the user community as standard FITS files. As we adopted ROOT relatively early, we have limited our direct use of ROOT within our software due to a desire to avoid tying ourselves too strongly to ROOT’s fortunes. Our science tools do make use of ROOT’s math libraries such as TMinuit. Fortunately, ROOT is very modular and while the framework has grown substantially over the years, we have been able to pick and choose those portions we want to use.

One of the primary advantages of ROOT is the excellent support provided by the ROOT team through its online documentation and forums. Response to our questions has been spectacular. While some complain about the trouble involved in producing publication quality plots, the python interface now makes that job a bit easier. ROOT has also started to offer CMake builds which is a nice improvement over the Cygwin build option on Windows.

We have tried to stay up to date with ROOT’s recommended production releases during our development phase. However, after launch, we have chosen stability over frequent upgrades, sometimes staying with a version of ROOT for 2 years or more. Despite this, upgrades of ROOT are relatively painless and easy to test.

6. A word about documentation

Creating the best software in the world is worthless without providing documentation to back it up. It is inevitable that users may run into obstacles that the original developers may not have expected. Initially we attempted multiple times to create useful documentation and keep it updated. Organization or lack thereof was typically the problem. Ultimately, we hired a professional technical writer, who worked with us for six years. Having no personal knowledge of how our software worked, he started from the basics and designed an online user workbook. The format of the workbook pages includes handy navigational toolbars that are now becoming more standard, but at the time were just entering the scene. This investment has paid off, as users and developers have an online resource that answers many of the typical questions that come up with using and writing the software.

7. Other tools

There were a number of other freely available tools we adopted up as well. The more notable choices were: Code Management Tool (CMT) [6] and later SCons [7], Doxygen [8], and EVO [9]. When we chose to support both Windows and Linux development, and then chose Gaudi as our framework, CMT became a natural choice. Gaudi uses CMT as its build tool, and we were reorganizing our software to match their package structure. Over time, CMT became a burden to the lack of developer support and need for customized handling for each version of Visual Studio. Ultimately, we chose to move to SCons as our build tool. This has not been an easy migration, but the resulting system is much easier to maintain and migrate to upcoming compilers and operating systems.

To facilitate communication, we use Doxygen for code documentation and EVO for all Fermi LAT team meetings. The Doxygen documentation format lends itself nicely to C++ programming. Doxygen syntax is easy to pick up and start using immediately. We adopted a uniform Doxygen format to be used across all the simulation, reconstruction, and analysis software. The documentation is automatically generated as part of our automated build process.

EVO has been a faithful tool to facilitate our meetings involving collaborators around the globe. Our EVO use has focused primarily on audio, rather than video or desk top sharing tools. When we have run into trouble on rare occasions, the EVO team has been very quick to help out.
7.1. Windows Support
We have a handful of proficient Windows developers who are attached to the Visual Studio development environment. Among their reasons, include: an integrated debugger and editor which allows quick navigation to problem areas of the code and the ability to adjust build options via a few clicks. While other tools are now available, it would be difficult to get our team to migrate at this point. Our user community, however, is primarily Linux based as is our production system.

As the project matures, and the number of developers available to support the maintenance of the software dwindles, it is becoming more difficult to maintain support for Windows. We have found that after launch, many of those who were contributing to the software effort are now focused on science analysis, and no longer have time to expend on software. Visual Studio itself continues to evolve with each release, and providing full Visual Studio project and solution files is a burden in CMT, which is a large part of the reason we migrated to SCons. However, even in SCons, we have had to provide python extensions to fully support the Visual Studio debugger as demanded by our developers. In retrospect, CMake may have offered a better choice in terms of windows support, despite its home grown scripting language.

8. Lessons Learned
External libraries are an extremely useful resource for common code that allows developers to avoid re-inventing the wheel. Fermi offline software was very fortunate to have access to software projects such as ROOT, Geant4, and Gaudi. Our development team could then focus its efforts on the tasks specific to Fermi.

Windows support continues to be an issue; both for our own development as well as for the external libraries we have chosen. Gaudi will likely drop Windows support entirely in the near future. Other externals have adopted CMake, which makes building on various platforms much easier.

When adopting any external library it is a very good idea to become active in its development community. These groups are a great resource when you run into trouble and have questions. We have found the communities of ROOT and Geant4 especially helpful.

Some libraries depend on yet other libraries. You may find that there are conflicting versions requirements across different external libraries. At best, upgrading one library, may force you to upgrade a number of others due to these dependencies. We have found this to be the case with Geant4 and Gaudi’s use of CLHEP. Future maintenance efforts will likely involve external upgrades. For this reason, it is a good idea to limit the number of external libraries utilized.

Upgrade as often as possible. It is much easier to handle incremental upgrades rather than jumping several versions at once. This is especially true of lower level externals such as a framework.

Over the lifetime of the mission, upgrades to modern operating systems and compilers occur every few years. Most of our external libraries have easily outpaced us in terms of supporting recent compilers. Others have lagged behind; this is particularly a problem on Windows.

Despite any hurdles encountered, using external libraries is worth any trouble or effort associated with them. This has certainly been the case for the Fermi LAT, where we have a small development team. Having freely available, quality software to turn to that takes care of the typical functions required for high energy physics projects is a great resource.

9. End Game
Now that Fermi is a mature project, it is time to prepare for the long term support of the data and software. It is not expected that we will continue to be able to port the offline software to future flavors of Linux and Windows and updated compilers indefinitely. Virtualization is now being considered as a path forward, where we can freeze our software on a particular compiler and operating system. While this work is still in its early stages, we are considering VirtualBox which provides support for the operating systems Fermi offline officially supports.
References

[1] Burnett T et al 2000 *Computing in Science & Engineering* **2** 9-18
[2] Agostinelli S et al 2003 *Nuclear Instr. And Methods A* **506** 250-303
[3] Brun R and Rademakers F Sep 1996, *Nucl. Inst & Meth. in Phys. Res. A* **389** 81-86
[4] Barrand G et al 2001 *Comput. Phys. Commun.* **140** 45-55
[5] Lonnblad L 1994 Comput. Phys. Commun. **84** 307-316
[6] CMT. (n.d.). Retrieved from http://www.cmtsite.org/
[7] SCons. (n.d.). Retrieved from http://www.scons.org/
[8] Doxygen. (n.d.). Retrieved from http://www.stack.nl/~dimitri/doxygen/
[9] EVO The Collaboration Network. (n.d.) Retrieved from http://evo.caltech.edu/evoGate/