Software Development, Integration and Distribution Tools in CMS

David J. Lange
Lawrence Livermore National Laboratory
E-mail: lange6@llnl.gov

Benedikt Hegner[*]
CERN
E-mail: Benedikt.Hegner@cern.gov

The offline software suite of the CMS experiment supports the production and analysis activities across a distributed computing environment. This system relies on over 100 external software packages and includes the developments of more than 250 active developers. This system requires consistent and rapid deployment of code releases, a stable code development platform, and efficient tools to enable code development and production work across the facilities utilized by the experiment. Recent developments have resulted in significant improvements in these areas. We report the concept, status, recent improvements and future plans for these aspects of the CMS offline software environment.

XII Advanced Computing and Analysis Techniques in Physics Research
November 3-7 2008
Erice, Italy

[*]Speaker.

1. Introduction

Software for the Compact Muon Solenoid (CMS) experiment must support a variety of user applications from a single, shared code base. Applications span the entire range of experimental work, and include the online trigger system, reconstruction and simulation executables, data quality monitoring, analysis-based event skimming and user data analysis. The requirements for these applications differ substantially, however a single code framework (CMSSW [1]) provides an efficient structure for code reuse and shared algorithm development. With hundreds of active developers, frequent software releases facilitate integration of new features as well as bug fixes identified in production activities.

In this paper, we discuss the development and status of the user software development environment, code integration system and distribution tools for the CMSSW software. CMS began the CMSSW project in 2005 and redeveloped substantial portions of its software release and development system at that time in order to support development of the CMSSW application code itself. Recently, these systems have undergone substantial development in order to streamline existing functionality and to expand a broader range of use cases.

2. CMSSW user environment

A single CVS repository is used to manage CMSSW software. Code is divided into approximately 1500 user defined packages. Each package can define one or more libraries. As discussed below, tagged versions of each of these packages are organized into release builds designed to meet collaboration development goals and timescales for production work. Developers use these release builds, and corresponding production event samples, to develop new capabilities or to provide bug fixes.

Figure 1 shows the growth in number of developers and source lines of code in CMSSW. During a typical month, more than 200 developers commit code for packages that are part of a CMSSW release. As one would expect, during the development phase of the CMS experiment, the code base has grown at a steady rate. We see almost linear growth through the beginning of 2009 in the primary coding language (C++). During 2008, the python language was adopted by the experiment as the new format for offline job control. As such, the amount of python code in CMSSW releases has grown considerably during the past year.

We have provided utilities to support developer interaction with these release builds. These include:

- showtags: Tracks packages under development and discovers changes with respect to underlying release build. This utility allows users to compare the CVS tags in their working area with those in the release itself as well as to compare (e.g., via cvs diff) their developed code with that in the release.
- addpkg: Query and checkout of CVS tags for each package in a given release using a command line interface (a web interface for package tag tracking is described below).
- checkdeps: Dependency checking to determine what packages must be recompiled to account for interface changes made within a developer's working area. Utility determines

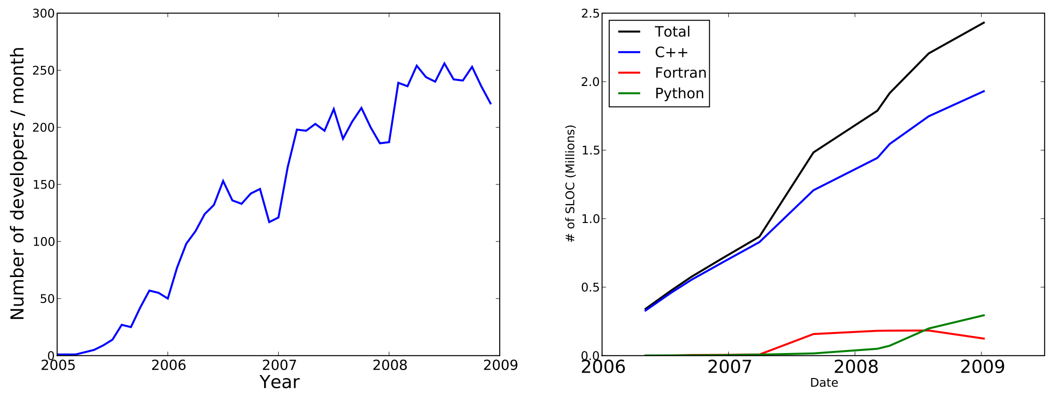

Figure 1: Changes with time of the number of CMSSW developers (left) and source lines of code (right).

which header files have changes, queries archived dependency information from the release build to determine which packages must be recompiled to account for the interface change and (optionally) checks out these packages.

CMS uses the SCRAM [2] package to define build rules, external dependencies and the software configuration. SCRAM is used both to perform full release builds and as an interface to "make" within a developer environment. SCRAM is maintained and developed within CMS and we have recently achieved significant improvements in the performance of SCRAM in user applications. Within SCRAM, each CMSSW release is defined as a project. Users can query SCRAM regarding available software releases at their site and then can choose the appropriate available software release for their development. SCRAM tracks the configuration of external packages required by each CMSSW project, as were defined by the release build system (described below).

CMSSW developers can use several independent project areas simultaneously, however the same developer area is not typically shared between developers. The tools discussed above are used to easily share CVS tagged source code between project areas or from developer to developer. For example, the addpkg utility supports checking out a list of package tags from a list specified in a file.

Each software package defines one or more "BuildFile"s that define the dependencies of that package. SCRAM generates "makefiles" and performs the library and binary product builds from these BuildFiles by determining the dependency ordering for each defined library and/or binary. In addition, SCRAM defines the run-time environment for CMSSW applications.

During the past year, we have improved the performance of SCRAM for generating makefiles and running make. The time to process packages within a developer area to determine appropriate makefiles to be used has been reduced by more than a factor of ten. In addition, the memory used in this process has been reduced by a factor of five. At the same time, the disk resident caches and generated makefile sizes were reduced by approximately a factor of ten. These improvements, in particular the reduction in time for processing BuildFiles, directly leads to a more efficient developer environment for CMSSW developers.

3. CMSSW Releases and Release Integration

Frequent releases of the entire CMSSW code base are an important aspect of the CMSSW project. The schedule and physics goals are established by the experiment as a whole and the CMSSW release schedule is established to meet these goals. Typically, goals are defined in terms of data taking periods or simulation production cycles. Three different types of release builds are created during each release cycle:

- “Integration builds”: twice per day, a full build of approved package tags is performed. Automated quality assurance and small runs of production workflows are performed to check the status of each nightly release. These releases are not distributed and have a lifetime of one week before being replaced.

- “Pre-releases”: Periodic builds that are distributed and used for code and physics performance validation during the initial portion of a release cycle when major features are under development.

- “Production releases”: Production releases represent releases ready for distribution fully across the grid and for use in major data taking and simulation sample production. Only production releases are used for these activities. There are typically a number of production releases within a single cycle to support bug fixes and to enable required features that were not ready for the initial production release.

Including both pre-releases and production releases, we have generated nearly 300 releases since 2007. Package tags for each of these builds are requested and tracked through a web-based tag collection tool [3]. This tool relies on a SQL database and archives user requests for new package tags as well as the set of CVS tags used to build each CMSSW release. In addition, we have defined a convention for allowed official CVS tags and prevent those package tags from changing once they are created.

A single person is typically responsible for release integration, updating both external software packages as well as CMSSW packages. We have adopted a hierarchical system, where coordinators of individual components are responsible for propagating the effects of interface changes to affected CMSSW code in other areas. Once requested, a package tag must be checked approved by the appropriate software coordinator. This extra requirement means that the nightly releases are typically fully functional. Initially, CMSSW was integrated using a completely open system, where the nightly builds suffered from frequent, extended, periods where the nightly releases had significant compilation problems, let alone runtime problems, due not properly accounted for dependencies between CMSSW packages.

4. CMSSW Release Builds and Distribution

To ensure consistent functionality, both in terms of performance and consistency of results, we create a CMSSW release build together with a fully consistent set of external package builds. CMSSW relies upon more than 100 external packages, which range from standard C++ libraries (e.g., gcc, boost) to generator packages and other software packages specific to high-energy physics applications (e.g., pythia and clhep).

We developed an integrated release and distribution system based upon the standard RPM and APT packages. We created RPM .spec files to define dependencies and build instructions for each external package as well as for CMSSW itself. While a substantial effort was required to define these build instructions, this system now ensures consistency of the CMS software run-time environment across the computing grid and allows us to quickly apply patches, even to external packages, if required.

To ensure consistency across CMSSW and its dependencies, our build system rebuilds all dependent packages whenever any .spec file is altered. While only required in certain instances, for example when an external interface is changed, our system removes the possibility that the release coordinator will incorrectly omit a needed package rebuild when creating a new release. This system is fully automatic and can build CMSSW and all of its external dependencies in about eight hours. In the typical case where no external packages have changed, CMSSW itself can be built in 4 hours (using a single node with eight processors).

During a typical release cycle, tens, or even a hundred or more, packages will often change between two releases. This is another demonstration of the significant development and validation within CMSSW. Once stabilized, often isolated code bugs (or unanticipated additional requirements) require a new release to support production efforts or data taking with a short turnaround requirement. While a full CMSSW build takes only a few hours, we have now developed the ability to build "patch" releases, which rebuild only the affected portion of a CMSSW release, depending on the already built production release for the majority of its functionality. These patch releases can be built in less than 30 minutes, once the package tags have been collected and verified.

Finally, we also distribute RPMs with only a portion of a full CMSSW release, corresponding to the core framework and data format packages as well as the packages required for online data taking (including the high-level trigger process). These RPMs primarily serve to reduce the disk space overhead for particularly important use cases where only a fraction of the entire CMSSW software is required. From a specified set of required packages (i.e., the specified required setup of functionality), we find the dependencies required, both CMSSW packages and external dependencies, to make a fully consistent build.

5. Conclusion and Future Work

We have discussed the mechanisms employed by the CMS collaboration to facilitate software development, release integration and software distribution. The CMSSW software structure has grown in its maturity, however development of physics algorithms still continues at a roughly constant rate.

We continue to develop tools to ease the burden of development within a large and changing software code base. Recent improvements have decreased the resources required to build and run make rules from user defined dependency information. With these improvements, the time required for building CMSSW code within a developer environment is limited by the time required to compile and build libraries itself, rather than by the creation of build rules and makefiles.

Development and production use of the CMSSW software depends on frequent code releases. We have developed a single build system to define and track code dependencies of CMSSW as well as its more than 100 external package dependencies. In this way, we automatically ensure that a consistent code base is available for distribution across the LHC grid sites.

Ongoing work focuses on speed and scaling issues as well as distribution across a wider set of platforms. In particular, we have identified limitations in RPM that limit the number of simultaneous releases that we can install. We now understand some of the reasons for this limitation and are looking at solutions that will be transparent to users. We believe that this will also reduce the time required to build CMSSW RPMs, which is one of the limiting steps towards achieving a further reduced build time and which is again related to the large CMSSW source tree. Finally, we have applied our build system to support MAC-OS builds of CMSSW or at least the subset of packages that corresponds to high-level physics analysis use case.

6. Acknowledgements

The authors acknowledge contributions and support from their CMS collaborators and in particular P. Elmer, G. Eulisse, S. Muzaffar, A. Pfeiffer and N. Ratnikova. D.Lange's work performed under the auspices of the U. S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

References

[1] The CMS Collaboration, CMS Physics Technical Design Report, Volumn 1: Detector Performance and Software, CERN-LHCC 2006-001.

[2] Tool for software configuration and management originally developed under the LCG software project

[3] The CmsTC interface, which is based on the WebSrt system developed by the *BABAR* collaboration to support CVS tag tracking for release integration.