Use of a Software Configuration Management Tool in LHCb

J.Harvey[1] *P.Mato*[1] *F.Ranjard*[1]

European Laboratory for Particle Physics (CERN), Genève, Switzerland

Abstract

LHCb Software is made up of a variety of software packages, including legacy code written in Fortran, a new software framework written in C++, as well as many externally developed packages. It is typical that each package is developed and maintained by a different person and therefore evolves independently of the other packages. A set of working and production areas has been set up to ensure that developers can release their packages at any time without disturbing the main development line. The software librarian maintains the packages in a CVS code repository, and a release procedure has been established that makes use of a special tool for building the LHCb data processing applications on the various supported platforms. Here we describe in detail the configuration management requirements that led to the adoption of CMT as our software release tool and our experience using this tool over a 1 year period.

Keywords: CMT, configuration management, tool

1 Introduction

The LHCb [1] experiment is supported by 50 institutes with more than 500 physicists from 15 countries. The current software is made of some legacy code, a set of packages written in Fortran, a new software framework [2] written in C++, as well as many packages not maintained by us. The software is developed on both UNIX and NT in the various institutes, is stored in a CVS [3] code repository, and runs on different platforms in different environments. A new version of a package may be released at any time and this release must be managed in such a way that the main development line is not disturbed. In addition, changes in language, compiler, and operating system must be envisaged as the software evolves.

These configuration management issues should be solved using a tool to relieve developers from the burden of writing makefiles that must take account of the various environments and platforms. At the same time the tool should help librarians with installation of new packages as well as specific versions of programs on different platforms. In addition, the developer should be allowed to customise the configuration of his own package and to query the configuration which has been used to build his application.

After evaluation of several existing **software release tools**, CMT [4] has been selected on the basis of its ability to satisfy these requirements and for its very convenient and intuitive user interface. CMT builds makefiles, which in turn are used to build libraries and executables. CMT reads a requirements file in order to determine which application has to be built and which compiler and link options are to be used. CMT can handle a single package or a collection of packages. Default options are available for compiler and link options. The user can query CMT in order to get the options used and can modify them.

2 Software Release Structure

A large software project, such as ours, spans many developers distributed over many geographic sites. The challenge is to partition the work such that pieces can be developed independently and then easily combined to form stable snapshots for each data processing application i.e. the reconstruction program, event display etc. The way in which LHCb software is physically managed therefore reflects the organisational structure of the LHCb project, as well as the logical structure of the code itself. A set of configuration management procedures has been introduced to support this distributed development process. The nomenclature used to describe these procedures follows that proposed by Lakos [5].

Related software components are combined into logically cohesive physical units, called **packages**. Each package is under the responsibility of a **package manager**, who is typically one of the main authors of the package. Packages are identified and retrieved by their name and version number.

A specific directory structure has been defined in order to facilitate the release procedure (Figure 1). Beneath the root directory of each package are sub-directories holding several parallel release structures (versions). Under the package's release directory are a number of sub-directories. The source sub-directory (**/src**) contains source code (C++, Fortran, HTML). The interface sub-directory (**/packagename**) contains include files accessible from other packages in the form #include "packagename/file.h". The documentation sub-directory (**/doc**) contains all documentation related to the package (ASCII, HTML,..). The manager sub-directory (**/mgr**) contains a **requirements** file which gives a high level description of the package environment, details on what to build (library, application) and how to build it (compile and link options), and where to find the constituents (file locations). There is a single requirements file for all platforms, including NT. In addition there are a number of subdirectories that contain binary products. Multiple versions of binary products are built according to the compiler and platform on which they run and the particular compile and link options that are available e.g. debug and optimisation level. The names of these binary directories are chosen to reflect these options.

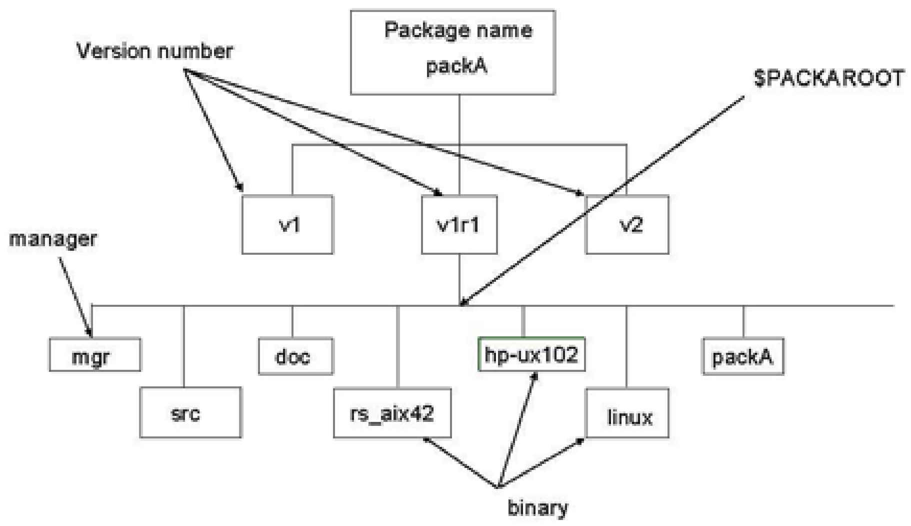

Figure 1: package layout

3 Package design issues

There are a number of physical design issues that can lead to problems at compile time, at link-time and at run-time. Many of these have been described in detail by Lakos [1] and have served as guiding principles when setting up our configuration management procedures. In particular dependencies between packages must be strictly controlled in order to minimise coupling between different software components. Minimising dependencies reduces the number of packages that must be linked and therefore the size of the executable image. Cyclic dependencies between packages can result in unresolved references at link time and can prevent staged releases. The existence of package dependencies is an architectural issue and must be controlled by the system architect at design time. Control can be exercised if dependencies are documented and validated by the librarian at build time. As a general rule, the number of exported header files should be minimised as the fewer details that are exposed in the interface of a package, the easier it is for a developer to maintain it.

Dependencies between packages can be declared to CMT in the package requirements file. CMT recognises dependencies at compile time and uses this information to rebuild libraries and executables accordingly. An example of a package requirements file is given below.

```
package packA
version v1
branches doc src mgr packA
include_dirs $(PACKAROOT)
use packB v1
use packR v2r1
library packA ../src/*.cpp
macro packA_linkopts ‘‘$(PACKAROOT)/$(packA_tag)/libpackA.a’’\
VisualC ‘‘$(PACKAROOT)/Win32Debug/packA.lib’’
```

Packages can be further categorised according to their special features:

- A **program** is a package which contains a **main** routine and a list of dependent packages needed to link it. Since the requirements file contains the version of the package and the name and version of all packages used by the application, it is easy to distribute a new version of the program, to archive it or to retrieve it.
- A **package group** contains a list of other packages with their version number valid for the specific version of the framework. To install the current version of the framework in a new environment it is sufficient to install the framework package and all dependent packages.
- An **external package** is a package that is developed and maintained by external groups. Familiar examples include CERNLIB, CLHEP, ROOT, XML, and GEANT4. These are normally released in binary form and their requirements file contains references to their interface and binary locations. The use of the CMTSITE environment variable allows the various locations to be defined in a single place.

4 Roles and Procedures

Essentially all LHCb people involved in software development use CMT, but the way in which each person uses it depends on the role they play. This section contains some examples illustrating this.

The **casual user** typically develops a software algorithm in his working area and builds an application by linking it with other selected packages from the public release area. The program package must firstly be checked out from the CVS repository with CMT. CMT will then build in the working area an image of the package. The user develops code in the **/src** area. The requirements file is modified to describe the application to be built. Finally gmake must be run from the **/mgr** area in order to create the application and this will be stored in the binary sub-directory corresponding to the compiler and platform option selected. A session that follows this sequence of commands would look as follows:

```
> cd somewhere
> cmt checkout LHCbprog
> cd LHCbprog/v1/src
...  add user code > cd ../mgr
> gmake
```

The **package developer** develops and maintains software for general public use and is expected to supply test routines and documentation in addition to the code. The developer will checkout with CMT the package he is working with, as well as the program package he wants to use to check it. CMT commands can be submitted from the **/mgr** sub-directory to check which packages will be used. CMT specifies default compiler and linker options, which can be queried and overridden if required. In the manager sub-directory belonging to the program package, CMT can be queried to get the location of used packages, and these locations can be modified if necessary by changing the CMT search path.

```
> cd somewhere
> setenv CMTPATH $PWD
> cmt checkout packA
> cd packA/v2/src
- modify some code
> cd ../mgr
> gmake
```

The **librarian** installs new versions of packages, programs or package groups in the public release area.This is achieved by checking it out of the repository with CMT in the public release area and then by running *gmake* from the **/mgr** sub-directory. CMT offers a special recursive mode to automatically checkout all packages that are dependent on the package being checked out, and also a broadcast facility to automatically build the corresponding libraries in their appropriate sub-directories.

To install a package group in a new site, the librarian will check-out the package recursively as for the program package. He will update the so-called external package requirements file to add the new site external library locations and then run CMT to configure and setup the new environment. From there he will use the CMT broadcast facility to build all libraries.

```
> cd $LHCBSOFT
> unsetenv CMTPATH
> cmt checkout -R LHCbprog
> cmt broadcast cmt config
> cd LHCbprog/v1/mgr
> cmt broadcast gmake
```

5 Experience with CMT

We have been using CMT for about one year both for the legacy Fortran code, which comprises 32 packages, and the new C++ Framework, comprising 10 packages. In addition we use some 10 external packages. Some of the Fortran packages are *used* by the C++ Framework. Our simulation program is maintained under CMT and is in production in 5 institutes in very different environments.

The management of our software has been improved by the combined usage of CMT and CVS. CMT is extremely simple and convenient to use and avoids the need to write complicated makefiles. Features, such as the query facility for identifying options used during build, as well as the possibility of inheriting and customizing build options from within the package hierachy, are heavily used. In addition it runs on NT and UNIX platforms which is an essential requirement for LHCb.

We conclude by saying that the use of a Configuration Management Tool from the very beginning of our software development activity has been essential for managing the contributions made by the various developers and for handling the various software development environments and platforms.

References

1 LHCb Technical proposal CERN/LHCC 98-4 LHCC/P4.
2 M.Cattaneo & al, “GAUDI - The Software Architecture and Framework for building LHCb Data Processing Applications”, CHEP2000, Padova, February 2000.
3 CVS free software.
4 C.Arnault, “CMT”, CHEP2000, Padova, February 2000.
5 J.Lakos, Large scale C++ Software Design, Addison Wesley, 1996