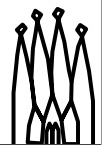
2

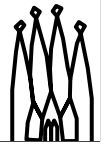
Configuration and Build System



Objectives

After completing this lesson, you should be able to:

- **Understand the LHCb Configuration Management**
- **Get a copy of a package from the LHCb code repository**
- **Know how to re-build libraries and programs**



Lesson Aim

Understanding and being able of using the basic commands of the configuration and build system is a pre-requisite for the rest of the Tutorial.

The commands introduced in this lesson would allow us to get a copy of the packages that we are going to use for the rest of the Tutorial.

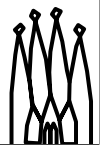
Caveat

The tools used in LHCb for configuring and building Gaudi programs are the same as for building Fortran programs (SICB), therefore people having already experience on those can skip this lesson.

Schedule:	**Timing**	**Topic**
	20 minutes	Lecture
	10 minutes	Practice
	30 minutes	Total

Package

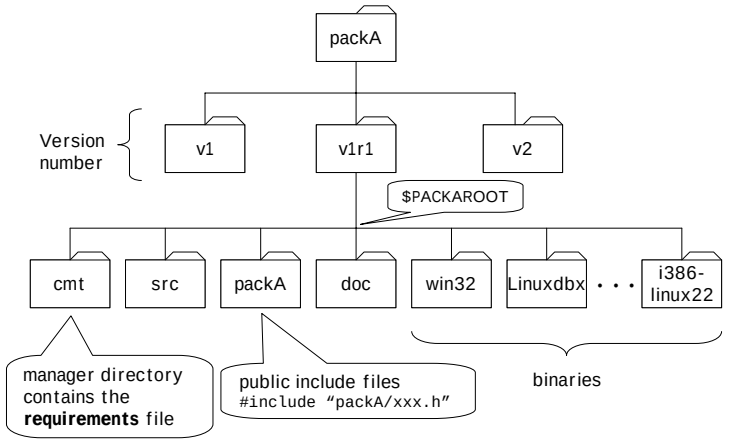
- **Package Definition**
  - **Collection of related classes in a logically cohesive physical unit**
  - **Minimal entity that can be versioned**
- **Reflects on**
  - **Logical structure of the application**
  - **Organizational structure development team**

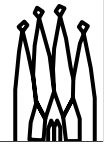


Package Definition

We define a “package” as a collection of related classes and as being the minimal entity that can be versioned and built. From this definition is clear that the configuration and build system we are using only deals with “packages” and not with individual classes or subroutines.

Package: Structure





Package Structure

All packages in LHCb follow the structure shown in the viewgraph.

- Package version name follow the convention “v<version number>r<release number>p<patch number>” or “v<version number>r<release number>d<date>”. Versions with the same v number should be considered compatible.
- The /cmt directory is mandatory (see later with CMT)
- The source files are located in /src and the exported header files are in /packA. In that way it is visible from which package a header is included.
- The /doc directory contains the documentation for the package. It is mandatory to have a file called “release.notes” where we keep the changes on the package up to date.
- A number of binary (platform & configuration dependent) directories will exists in each package.

Package Groups (Hats)

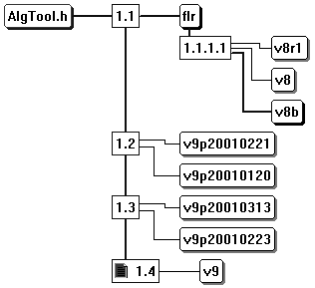
We have organized the packages in LHCb by putting together all related packages into a package group. Examples are: Event, Det, SICB, Ex, etc.

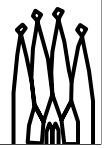
CVS

Version Control System

- **Record the history of your source files**
- **Helps you if you are part of a group of people working on the same project.**

(Repository, Module, File, Version, Tag)





Basic Description of CVS

CVS is a system that lets groups of people work simultaneously on groups of files (for instance packages).

It works by holding a central `repository' of the most recent version of the files. You may at any time create a personal copy of these files by `checking out' the files from the repository into one of your directories. If at a later date newer versions of the files are put in the repository, you can `update' your copy.

You may edit your copy of the files freely. If new versions of the files have been put in the repository in the meantime, doing an update merges the changes in the central copy into your copy.

When you are satisfied with the changes you have made in your copy of the files, you can `commit' them into the central repository.

When you are finally done with your personal copy of the files, you can `release' them and then remove them

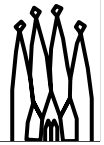
CVS: Common Repository

- **LHCb Code Repository**
  - **/afs/cern.ch/lhcb/software/CVS**
- **$CVSROOT**
  - **Using AFS**

    CVSROOT = /afs/cern.ch/lhcb/software/CVS
  - **Using CVS SERVER**

    CVSROOT =
    :pserver:cerncvs@lhcbcvs.cern.ch:/local/newlhcbcvs

    Password: CERNuser (read only)



CVS Repository

Code Repository Location

The LHCb code repository is located in the AFS volume **/afs/cern.ch/lhcb/software/CVS** with access restricted to the LHCb collaboration members.

For systems with direct connection to AFS, the CVSROOT can be defined to directly access to the repository using your local authentication
CVSROOT = /afs/cern.ch/lhcb/cvs/cvsmaster

Using the CVS server

If AFS is not available in your platform, then the LHCb CVS repository can be accessed using the cvs server. You have to specify the following option in you cvs command:

CVSROOT = :pserver:cerncvs@lhcbcvs.cern.ch:/local/newlhcbcvs

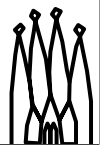
You have to login to the cvs server first. The server will ask for a password, reply CERNuser. You can now send all the cvs commands that don't require write access. If you use a command like **commit**, you will get an error message. When you have finished, you can logout of the server.

If you require write access to the repository, you will need a personal account and its corresponding password. Make your request to lhcb-helpdesk@cern.ch.

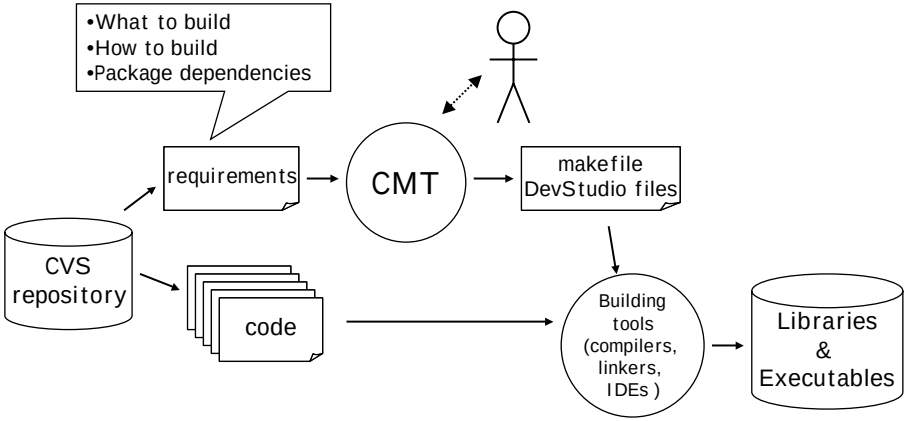
CMT

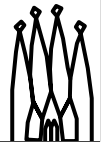
Configuration Management Tool written by C. Arnault (LAL, Orsay)

- **It is based around the notion of *Package***
- **Provides a set of *tools for automating* the configuration and building packages**
- **It has been adopted by LHCb (other experiments are also using it)**



How we use CMT





CMT introduction

CMT is an attempt to formalize software production and especially configuration management around a *package*-oriented principle. The notion of *packages* represents hereafter a set of software components (that may be applications, libraries, documents, tools etc...) that are to be used for producing a *system* or a *framework*. In such an environment, several persons are assumed to participate in the development and the components themselves are either independent or related to each other.

The environment provides conventions (for *naming* packages, files, directories and for *addressing* them) and tools for *automating* as much as possible the implementation of these conventions. It permits to *describe* the configuration requirements and automatically deduce from the description the effective set of configuration parameters needed to operate the packages (typically for *building* them or *using* them).

How we use CMT

The user interacts mainly with the CMT tools to configure and build the packages. The instructions on what to build, how to build and dependencies are located in a single text file called *requirements*. Very often the user needs to edit this file.

From the *requirements*, CMT is able to automate the creation of the makefiles (or Visual Studio projects) required for building the different package constitues (libraries, programs, documentation, etc).

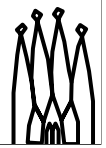
CMT: Requirements file

```
package Main
version v4r0
branches doc src job options cmt

use Components    v5r0 Tutorial
use GaudiConf     v6*
use GaudiSys      v9*
use LHCbEvent     v12*  Event
use DbCnv         v5*  Event
use GaudiRootDb   v5*
use HbookCnv      v11*
use SicbCnv       v12* SICB
use dbase         v234 SICB

#==> Main Program
application Main ../src/GaudiMain.cpp

# ================== set macros ========
ignore_pattern package_stamps
macro Main_linkopts  "$(application_linkopts)"
```



Some basic keywords

•**Package**. Defines the name of the package

•**Version**. Defines the version of the package (version naming conventions)

•**Use**. Instructs CMT on the dependencies of this package to other CMT packages. The wildcard "*" is allowed but should be used with care to avoid using incompatible versions, which may make the program to crash or produce invalid results. The name after the version is the location of the package (package group).

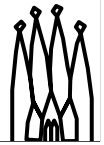
•**Application**. Tells CMT that this package is constituted of a program (application) and where to locate the sources.

•**Ignore_pattern**. Tells CMT to ignore a "global" pattern (typically a macro definition) that is convenient to have it defined for most of the packages. In this case this is done because the package is an application and not a library (majority).

•**Macro**. Defines a macro. In this case the macro Main_linkopts is used in the link statement for building the application.

CMT: Environment Variables

- **CMTPATH**
  - **Directories to look for CMT packages in addition to ones in the "~/.cmtrc"**
- **CMTSITE**
  - **Defines in which "site" you are. Used for selecting the adequate macro/environment variable depending on the site.**



CMT Environment Variables

•CMTPATH
Is the list of top directories to look for CMT packages. In LHCb environment is set at login time to:
/afs/cern.ch/user/<u>/<user>/newmycmt
Eventually you can add the DEV area into the path
/afs/cern.ch/user/<u>/<user>/newmycmt:/afs/cern.ch/lhcb/software/DEV
For the tutorial we will be using:
/afs/cern.ch/user/<u>/<user>/tutorial:/afs/cern.ch/lhcb/software/DEV

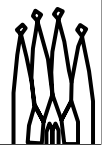
•CMTSITE
Used to select definitions of macros or environment variables (sets) according to the user's site. For example this is a fragment of the requirements file for ExternalLibs:

```
set SWROOT      "${SITEROOT}/sw"\
    NIKHEF      "/project"\
    CCIN2P3     "${LHCBHOME}/software"\
    CPPM        "${LHCBHOME}/software"\
    LAPE_RIO    "${LHCBHOME}/software\
    LBNL        "/auto/atlas"
```

•CMTROOT
Location of the CMT installation

CMT: Basic Commands

- **cmt config**
  - **Configures the package**
  - **Sets environment (source setup.csh)**
- **cmt show uses**
  - **Show dependencies and actual versions used**
- **cmt show macro <macro>**
  - **Show the value of a macro for the current configuration**



CMT Primary commands

•configure a package

> cd mycmt/package/v1/cmt

> cmt config

> source setup.csh

•visualize packages and version numbers used by a library or an application.

> cd mycmt/package/v1/cmt

> cmt show uses

•get macro definitions used in makefiles

> cd mycmt/package/v1/cmt

> cmt show macros

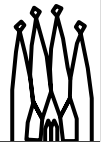
•get one specific macro used in makefiles

> cd mycmt/package/v1/cmt

> cmt show macro fflags

Package Categories

- ***Program*: is a package that contains a main routine and a list of dependent packages needed to link it.**
- ***Library*: contains a list of classes and the list of dependent packages needed to compile it.**
- ***Package group*: contains a list of other packages with their version number (e.g. GaudiSys)**
- ***Interface package*: interfacing to packages not managed with CMT (e.g. CERNLIB, CLHEP, ROOT,…)**



Package Categories

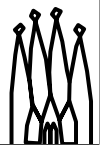
With respect to CMT it is interesting to distinguish the different categories of packages. They are used in the exactly the same way but their requirement files will show same differences, specially in patterns that are used.

The concept of **interface package** is interesting for integrating in the build system packages that have been developed outside CMT. Basically these packages only defines a number of macros and environment variables needed for compiling, linking and running with this external packages.

Package group are useful for fixing a set on compatible versions of other packages. In this case the package using this set of packages needs only to state the version number of the package group.

Link vs. *Component* Libraries

- **Link libraries are need for linking the program (static or dynamic)**
  - **Traditional libraries.**
- **Component libraries are loaded at run-time (*ApplicationMgr.DLLs* property)**
  - **Collection of components (Algorithms, Converters, Services, etc.)**
  - **Plug-in**



Component Libraries

Component libraries are shared libraries that contain standard framework components which implement abstract interfaces. Such components are Algorithms, Auditors, Services, Tools and Converters. These libraries do not export their symbols apart from the one which is used by the framework to discover what components are contained in the library.

The Tutorial will be based on the development of a “component” library that will include all the Algorithms that we are going to develop during the practical exercises.

Component Libraries

Components_load.cpp

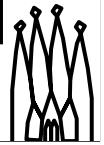
```
#include “GaudiKernel/DeclareFactoryEntries.h”
DECLARE_FACTORY_ENTRIES ( Components ) {
  DECLARE_ALGORITH( MyAlgorithm )
  DECLARE_SERVICE( MyService )
}
```

Your components need to be added here

Components_dll.cpp

```
#include “GaudiKernel/LoadFactoryEntries.h”
LOAD_FACTORY_ENTRIES ( Components )
```

No change needed



Component Libraries

In order to satisfy the requirements of a component library, two additional files must also be present in the package. One is used to declare the components, the other to load them. Because of the technical limitations inherent in the use of shared libraries, it is important that these two files remain separate, and that no attempt is made to combine their contents into a single file.

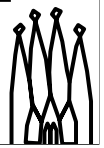
<Components>_load.cpp This file contain the declaration of all the components that should be available in the component library. There have been same macros defined to ease the writing of this file.

<Components>_dll.cpp This file is fixed and needs to be added into the package and do not need to be updated if new components are added later into the package.

Getting a package

- The “getpack” command
  - Script combining “cvs checkout” + “cmt config”
  - It suggest the latest version of package

```
> getpack [hat/]<package> [<version>] [head]
```



Getting a package

The “getpack” command has been developed to ease the use for getting a copy of a package from the LHCb cvs repository and putting it in the correct directory structure with the correct version. It also suggest the user what versions are available for the package in case the user does specify it.

For the tutorial we recommend to get always the “head” revision of the packages.

Building a package

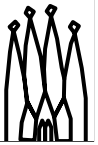
- Working in the /cmt directory
  - <package>/<version>/cmt
- Set correct environment

```
> source setup.csh [-tag=<configuration>]
```

- The gmake command

```
> gmake [target] [tag=<configuration>] [clean]

        configurations:   rh61_gcc2952 (default)
                          rh61_gcc2952dbx (for debug)
```



Building a package

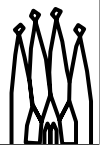
Typically for building (and also running) a package we stay in the /cmt directory. This is convenient for executing the configuration and build commands.

•**Setting the environment**. We need to setup the correct environment for the current version of the package. This environment consists on a set of environment variables (PATH, LD_LIBRARY_PATH, and others). Setting the environment is done my executing “source setup.csh” in the .cmt directory. This must be done before running the application always and in some cases also is needed for building the package. Therefore, we suggest to do it before building the package. You only need to re-do the setting of the environment if you change the package or the version you are using otherwise the environment stays valid for the complete session. We suggest for the tutorial to use the “debug” configuration by issuing the command “source setup.csh –tag=$CMTDEB”

•**The gmake command**. The “gmake” command is used for building the libraries and/or the programs. There are various configurations available in form of “tag=<configuration>”. We suggest for the tutorial to use the “debug” configuration by issuing always the command “gmake tag=$CMTDEB” since the default is without debug information.

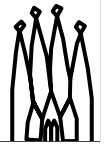
Tutorial Packages

- **Tutorial/Main [v4r0]**
  - **The *main program*. During the tutorial will be using the same program.**
- **Tutorial/Components [v5r0]**
  - **A package consisting of a single *Component* library in which we will be adding all the Algorithms of the Tutorial**



Exercise

- **Get both Tutorial Packages [v4r0] and build them**
  - **Remember to "source $LHCBHOME/scripts/tutorial.csh**
  - **Remember to build first the dependent package (Components) and then the program (Main)**
- **Execute the Main program**
  - **It should do nothing for the time being**



Tutorial/Main

This is the main program. We will use it without modification during the tutorial.

- /cmt/requirements requirements file
- /src/GaudiMain.cpp main program
- /options/jobOptions.opts job options file to be used during the tutorial (everything is commented to start with)

Tutorial/Components

This is the component library. We will populate the /src directory with new files from other /src.<xxxx> which contains the solutions to the different exercises of the tutorial.

- /cmt/requirements requirements file
- /options/<exercise>.opts job options "fragments" for the different exercises
- /src/Component_dll.cpp Needed for building a "component library"
- /src/Component_load.cpp Needed for building a "component library"
- /src/DecayTreeAlgorithm.* "Empty" .h and .cpp files for the Decay Tree exercise
- /src/VisibleEnergyAlgorithm.* "Empty" .h and .cpp files for the Visible Energy exercise
- /src/EventSummaryCnv.cpp "Empty" file for the Event Summary exercise
- /src.decaytree/* Directory with solutions for the Decay Tree exercise
- /src.hist_tuple/* Directory with solutions for the Histogram Ntuple exercise
- /src.solution/* Directory with the final solution

Exercise

```
> source $LHCBHOME/scripts/tutorial.csh
> cd ~/tutorial
> getpack Tutorial/Main v4r0
> getpack Tutorial/Components v5r0

> cd Tutorial/Components/v5r0/cmt
> gmake tag=$CMTDEB

> cd ../../../Main/v4r0/cmt
> source setup.csh –tag=$CMTDEB
> gmake tag=$CMTDEB

> ../$CMTDEB/Main.exe ../options/jobOptions.opts
```