Getting started with the D0 (analysis) software

Marco Verzocchi
University of Maryland

Goals:

- introduction to D0 software
- use simple english as much as possible, assume some C++
- start with an example, explain how it works along the way
- understand how you can write a program inside the D0 framework
- accessing physics quantities for analysis
- this should enable you to write simple analysis code
- build some expertise/understanding in the working groups

at the same time

- understand how other programs work (reconstruction, simulation, L3 triggers, online monitoring)
- starting point for getting involved in algorithms development and software documentation

this afternoon

- Heidi will deal with data access (SAM) this afternoon
- Laurent will show you how to use the event display

Disclaimer

- You may think that I'm a SW expert: I'M NOT
  I still have problems using * and &
- I may still not understand correctly some of the things I will be discussing
- It has taken me a lot of time to understand certain things
- Hopefully this tutorial will serve as an incentive for an improvement of the documentation
- I'll try to use simple examples and plain english as much as possible
- Please don't come to me asking very complicated questions later...

Contents

part 1 (8:15-9:00)

- D0 software black magic explained (those setup commands)
- Software versioning and CVS
- Understanding the directory structure of different packages
- How to build an executable
- The D0 software framework
- Framework RCP

part 2 (9:05-9:50)

- More on run control parameters (RCP)
- Understanding the RCP databases, problems with RCP files
- How to share data between packages (the EDM)
- What is the purpose of all those interfaces
- Event filtering
- Input/output
- Do and don't with the gmake command
- Using the d0tools to run D0 programs

Contents

part 3 (10:20-11:05)

- The D0ChunkAnalyze example
- Accessing some D0 physics objects from the chunks
- Writing "Elvis has just left the building" and Elvis is dead"
- Filling histograms and ROOT tuples

part 4 (11:10-11:50)

- Other chunks
- Chunk documentation
- Trigger selection
- Stuff I wanted to cover, but didn't find time for it:
  - RTE
  - d0cuts
  - luminosity calculation and bookkeeping
  - np_tmb_stream
- Documentation
- Yes you can contribute.....

Let's start

```
 ____________________________________________________________________
/                                                                    \
| Welcome to CLuED0 (Clustered LinUx Environment at D0)              |
|                                                                    |
| Node:   cole-clued0                                                |
| Owner:  University of Maryland                                     |
| Location:  PK177-37                                                |
| Subcluster:  green                                                 |
|                                                                    |
| Documentation is available at: http://www-clued0.fnal.gov/clued0   |
| D0 software release installed: type 'd0rel' to see which releases  |
| firewall activated: only www-clued0 will be visible from outside FNAL |
|                                                        D0 09/10/01 |
\____________________________________________________________________/
[31] mverzocc@cole-clued0:/home/mverzocc>cd /work/cole-clued0/mverzocc
[32] mverzocc@cole-clued0:/work/cole-clued0/mverzocc>setup D0RunII p12.03.00
**** setup D0RunII p12.03.00
[33] mverzocc@cole-clued0:/work/cole-clued0/mverzocc>setup d0cvs
[34] mverzocc@cole-clued0:/work/cole-clued0/mverzocc>newrel -t p12.03.00 EXAMPLE
Creating a test release "EXAMPLE" in the directory
    /work/cole-clued0/mverzocc
[35] mverzocc@cole-clued0:/work/cole-clued0/mverzocc>cd EXAMPLE
[36] mverzocc@cole-clued0:/work/cole-clued0/mverzocc/EXAMPLE>
```

Things to learn from this screen dump:

setup n32

setup D0RunII p12.03.00

setup d0cvs

newrel -t p12.03.00 EXAMPLE

What is setup ?

setup is a Fermilab addition to UNIX (and NT?). Makes sure that you can use a given product. Example (on d0mino):

$> which pine

pine: Command not found

$> setup pine

$> which pine

/usr/products/pine/IRIX-6/v4_21/bin/pine

Ok now you can read/send E-mails. What setup really does is to change your PATH variable so that the system knows where to look for pine...

Notice the IRIX-6 ? It allows the system to decide which machine you are on (automatically deals with different architectures). And the v4_21 ? It tells the system to use version v4_21 of pine.....

Use it also for the D0 software. Tons and tons of environmental variables. Just type *env* before and after doing *setup XXXXX*....

Using the setup command:

- **interactively**
  the setup command is defined during the login process

- **in batch (d0mino and CAB)**
  add to your shell script the following line
  . /usr/local/etc/setups.sh

- **in batch (on clued0)**
  add to your shell script the following line
  . /fnal/ups/etc/setups.sh

By the way in the following I will be always using the tcsh interactively. Use /bin/sh or /bin/bash for scripts running in the batch system !

setup n32: not in the screen dump, because I'm working on clued0 (Linux), needed on d0mino (IRIX), because d0mino can work either as a 64 bit machine or be restricted to 32 bits. Tells the system to get the versions of the code and of the libraries compiled with a certain (-n32) switch....

setup D0RunII p13.00.00 (I have p12.03.00 on the slide, because p13 didn't exist at the time I wrote the slides): choose the version of the D0 software you'll be using to compile/run your code

setup d0cvs: tell the system that you want to use CVS and that you want to get the software out of the D0 repository (and not out of the MINOS one)

newrel -t p13.00.00 EXAMPLE: create a working area for release p13.00.00 and call it EXAMPLE. You'll be doing all the work inside the EXAMPLE directory

Introducing the concept of D0 software version.....

(http://www-d0.fnal.gov/computing/algorithms/howto/howtousereleases.html)

- **D0 software: > 1000 different packages (see next slides)**
- **Ensure they all work and that they all work together**
- **Try to compile new version every week (t02.36.00 this week): t stands for test (development)**
- **Every 3 months make a production (p) release (p13.00.00 this week): production releases are used for official reconstruction and simulation on the farms.**
- **Working with private/unofficial releases for anything beyond developing and testing software is EVIL**

pxx.yy.zz:

xx is the version number (change every 3 months)

yy changes whenever bug fixes are introduced

zz changes for very small bug fixes or RCP changes

What does setup D0RunII p13.00.00 do in reality ?

Create ~300 environmental variables, controlling:

- **which version of the compilers is used**
- **which version of the system libraries is used**
- **which version of the D0 libraries is used (*$SRT..... variables*)**
- **how a program will be compiled**
- **which version of external SW programs is used (SAM, ROOT, Pythia, Herwig, CERN libraries, Python, Perl, TCL/TK, SQL,.....)**
- **which machine is being used (*$BFARCH*)**
- **where the RCP database is located**
- **how to access the databases**

Overriding these environmental variables is not recommended

setup d0cvs: what is CVS ? (1)

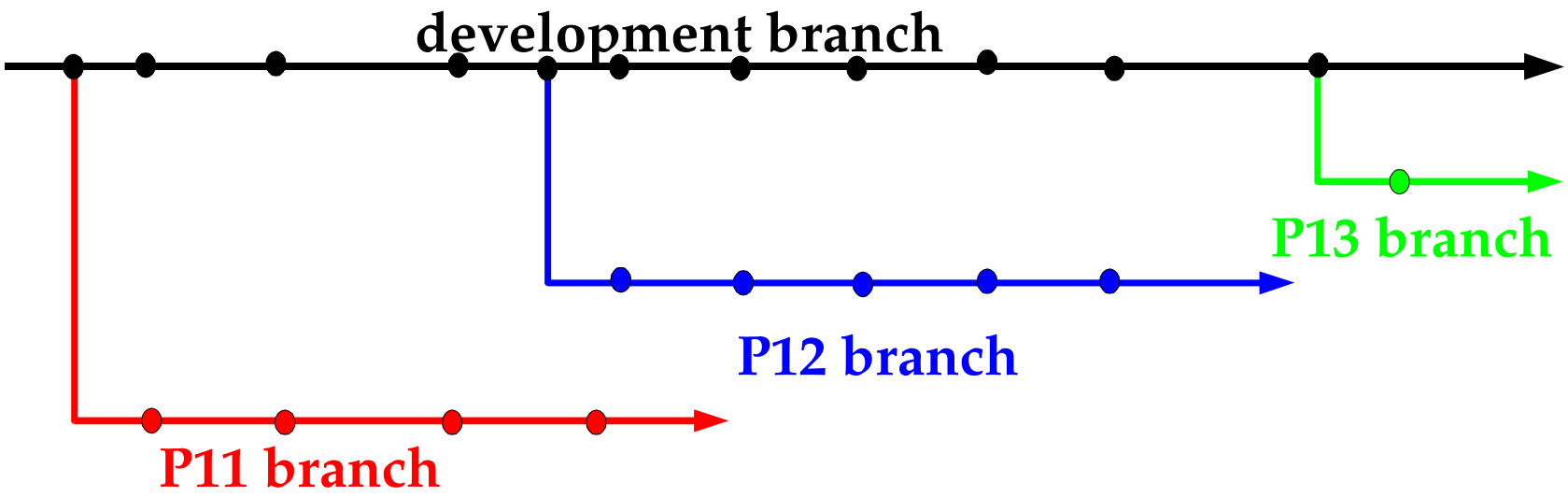
- Keep track of modifications to code, make different versions of code available on request
- Ensure that several developers can simultaneously change the code (and help resolve conflicts)
- Almost a standard in SW world

How does it work ?

- Software kept in a repository
- Users do not access (edit) files in the repository directly
- Get a local copy, then eventually put it back after modifications (resolve conflicts at this stage)
- Each change is logged. Every time a file is changed it gets an internal version tag

setup d0cvs: what is CVS ? (2)

- Collections of files are called packages. Assign a version number to packages (tag)
- CVS knows that when you say t02.36.00 you want a collection of packages, each with it's own tag
- When a production release is created, a branch is made in CVS
- Sometimes need to make changes both in the development and in the production branch



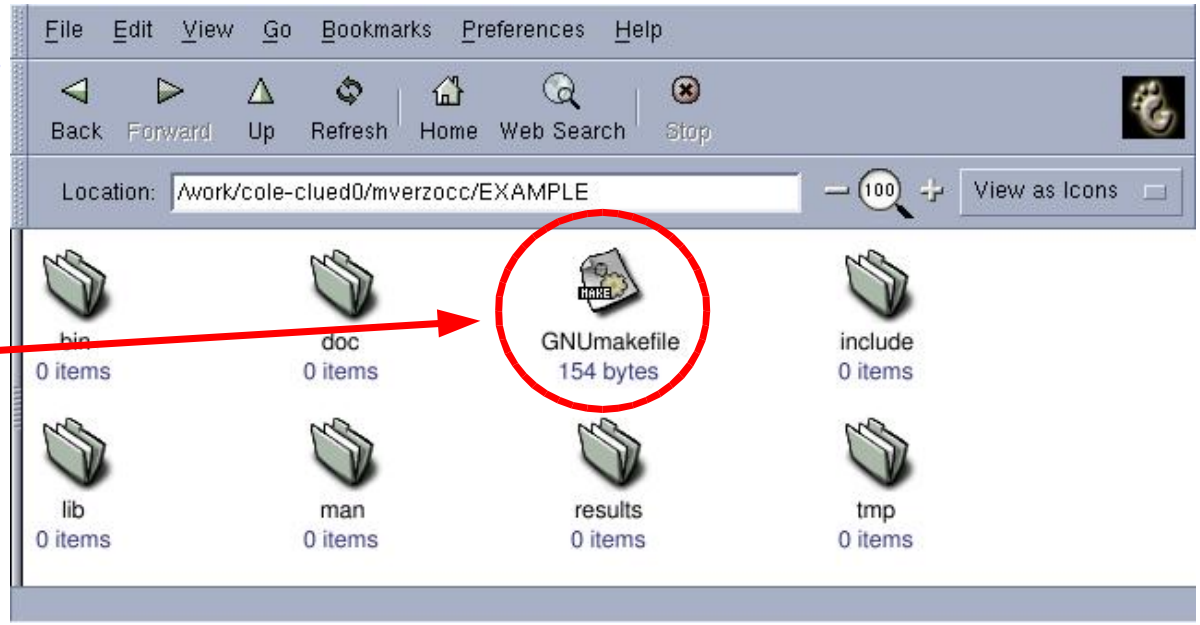
What does setup d0cvs do?

- **Changes the PATH variable so the system knows where CVS and related commands are**
- **Sets the CVSROOT variable (where is the repository)**

Almost ready for the next step: newrel -t p13.00.00 EXAMPLE:

- **Creates a working area with the appropriate structure for the D0 code building mechanism (SRT)**

An empty directory structure is created



DO NOT CHANGE the GNUmakefile

Using CVS: *(http://www.cvshome.org/docs)*

- **cvs help** — prints help page
- **cvs checkout** — extracts code (file/directory) from the repository
- **cvs add** — marks a file/directory for addition into the repository
- **cvs remove** — marks a file/directory for removal from the repository (files are never really removed)
- **cvs update** — update local copy with eventual changes done (by somebody else) in the repository
- **cvs commit** — really makes the changes in the repository

always run these commands first with the -nq options to see what will actually happen and make sure it's want you want to do

- **cvs diff** — find out differences between different CVS trees

---

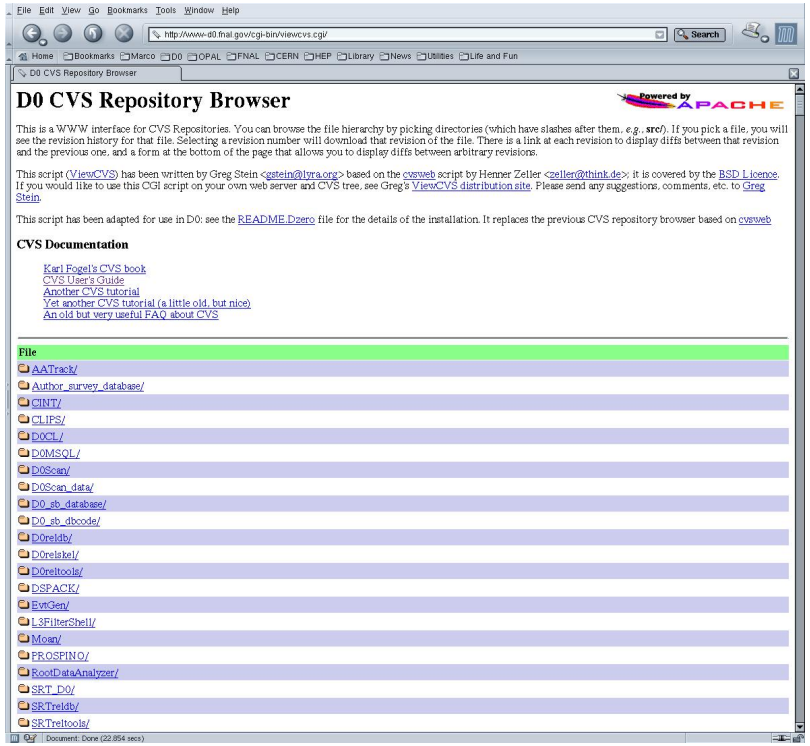
addpkg — does a cvs checkout of an entire package, create appropriate links for the D0 build mechanism

Browsing the CVS repository

(looking for differences between different versions of the code)

(*http://www-d0.fnal.gov/cgi-bin/viewcvs.cgi*)

- Very easy to use
- Allows you to track changes between different releases
- A little bit slow at startup



Other ways of browsing code:

http://www-d0.fnal.gov/D0code

- Cross references between packages (only for development branch, built daily)

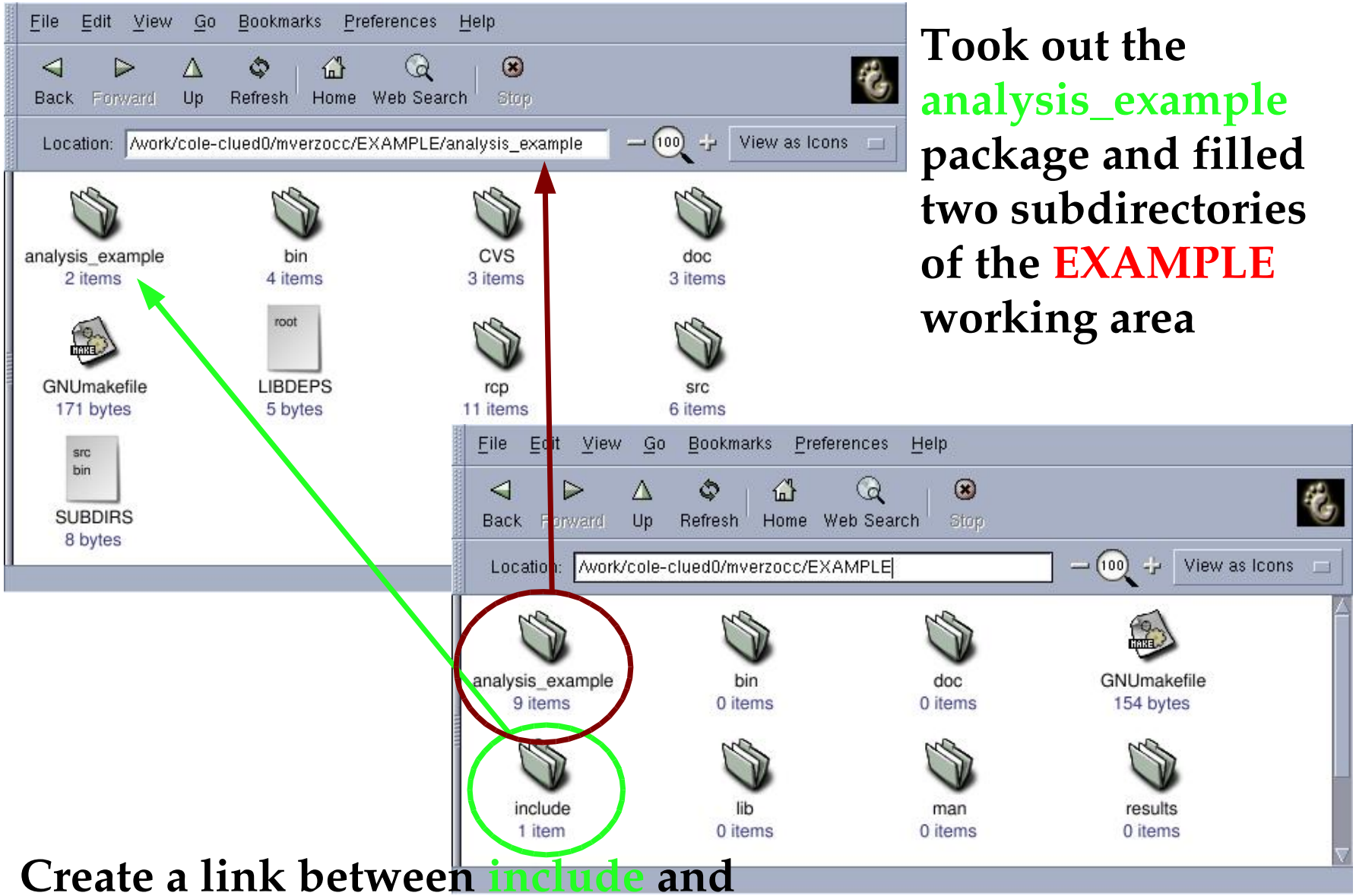
http://www-d0.fnal.gov/d0dist/dist/packages
http://www-d0.fnal.gov/d0dist/dist/releases

- Plain copy of /d0dist/dist disk area (good way of accessing the documentation directly, in the doc directory of each package)

Ok back to the screen and to the example

```
[35] mverzocc@cole-clued0:/work/cole-clued0/mverzocc>cd EXAMPLE
[36] mverzocc@cole-clued0:/work/cole-clued0/mverzocc/EXAMPLE>addpkg -h analysis_example
Adding package "analysis_example" to ".".
cvs checkout -P analysis_example
cvs server: Updating analysis_example
U analysis_example/GNUmakefile
U analysis_example/LIBDEPS
U analysis_example/SUBDIRS
cvs server: Updating analysis_example/analysis_example
U analysis_example/analysis_example/D0ChunkAnalyze.hpp
cvs server: Updating analysis_example/analysis_example/analysis_example
cvs server: Updating analysis_example/analysis_example/bin
cvs server: Updating analysis_example/analysis_example/doc
cvs server: Updating analysis_example/analysis_example/rcp
cvs server: Updating analysis_example/analysis_example/src
cvs server: Updating analysis_example/analysis_example/src/d0chunk_analyze
cvs server: Updating analysis_example/analysis_example/src/utilities
cvs server: Updating analysis_example/bin
U analysis_example/bin/BINARIES
U analysis_example/bin/D0ChunkAnalyze_x.cpp
U analysis_example/bin/OBJECTS
cvs server: Updating analysis_example/doc
U analysis_example/doc/index.html
U analysis_example/doc/tutorial.pdf
cvs server: Updating analysis_example/rcp
U analysis_example/rcp/D0ChunkAnalyze.rcp
U analysis_example/rcp/DropChunks_recreateTMB.rcp
U analysis_example/rcp/DumpEvent.rcp
U analysis_example/rcp/WriteEventDST.rcp
U analysis_example/rcp/WriteEventTMB.rcp
U analysis_example/rcp/runD0ChunkAnalyzeSAM_DST.rcp
U analysis_example/rcp/runD0ChunkAnalyzeSAM_TMB.rcp
```

I did cut the output



Took out the analysis_example package and filled two subdirectories of the EXAMPLE working area

Create a link between include and analysis_example/analysis_example

We'll learn a lot by understanding what is in the analysis_example directory.....

```
cvs server: Updating analysis_example/rcpdb/Linux2.4-KCC_4_0/personal/000/000/000
cvs server: Updating analysis_example/rundata
cvs server: Updating analysis_example/rundata/trigconfig
cvs server: Updating analysis_example/src
U analysis_example/src/COMPONENTS
U analysis_example/src/D0ChunkAnalyze.cpp
U analysis_example/src/D0ChunkAnalyze_t.cpp
U analysis_example/src/OBJECT_COMPONENTS
U analysis_example/src/RegD0ChunkAnalyze.cpp
cvs server: Updating analysis_example/src/d0chunk_analyze
cvs server: Updating analysis_example/src/utilities
[37] mverzocc@cole-clued0:/work/cole-clued0/mverzocc/EXAMPLE>srt_setup SRT_QUAL=maxopt
[38] mverzocc@cole-clued0:/work/cole-clued0/mverzocc/EXAMPLE>d0setwa
[39] mverzocc@cole-clued0:/work/cole-clued0/mverzocc/EXAMPLE>gmake all >& gmake.log
```

In the meantime we can compile the code (it takes some time):

srt_setup SRT_QUAL=maxopt **:**

tells the system to use the optimized version of the code instead of the version with the debugger information

(*srt_setup SRT_QUAL=default* to switch back)

Gain in compilation time and in running time (2x)

Use it as much as possible

- *srt_setup SRT_QUAL=.....* **changes also some of the environmental variables: the executable will not go into**

bin/Linux-2.4-KCC_4_0/....

but in

bin/Linux-2.4-KCC_4_0-maxopt/....

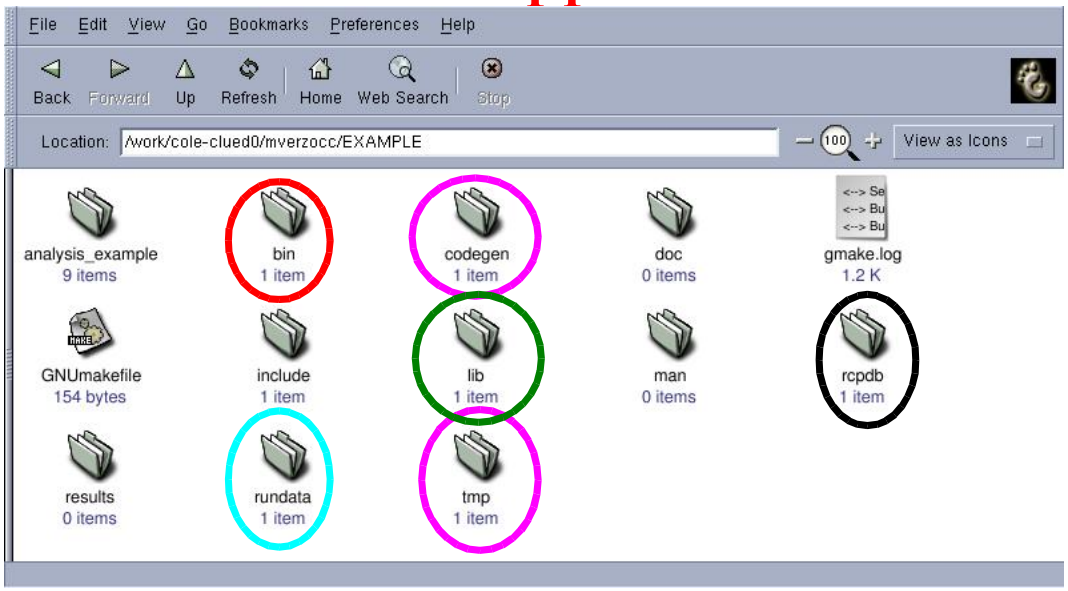
- **It also tells the system to pick up the optimized version of all libraries**

You should also do *d0setwa* **at this stage:**

- **it changes the environmental variables controlling where the RCP are (absolutely needed later at runtime)**
- **it changes the path for finding libraries and include files (from ./.... to /work/cole-clued0/mverzocc/EXAMPLE/.....)**
- **You really need it if you are making changes in the programs used to build the D0 software. You really need it to run tests (*gmake test*)**

And finally *gmake all* **to build the executable**

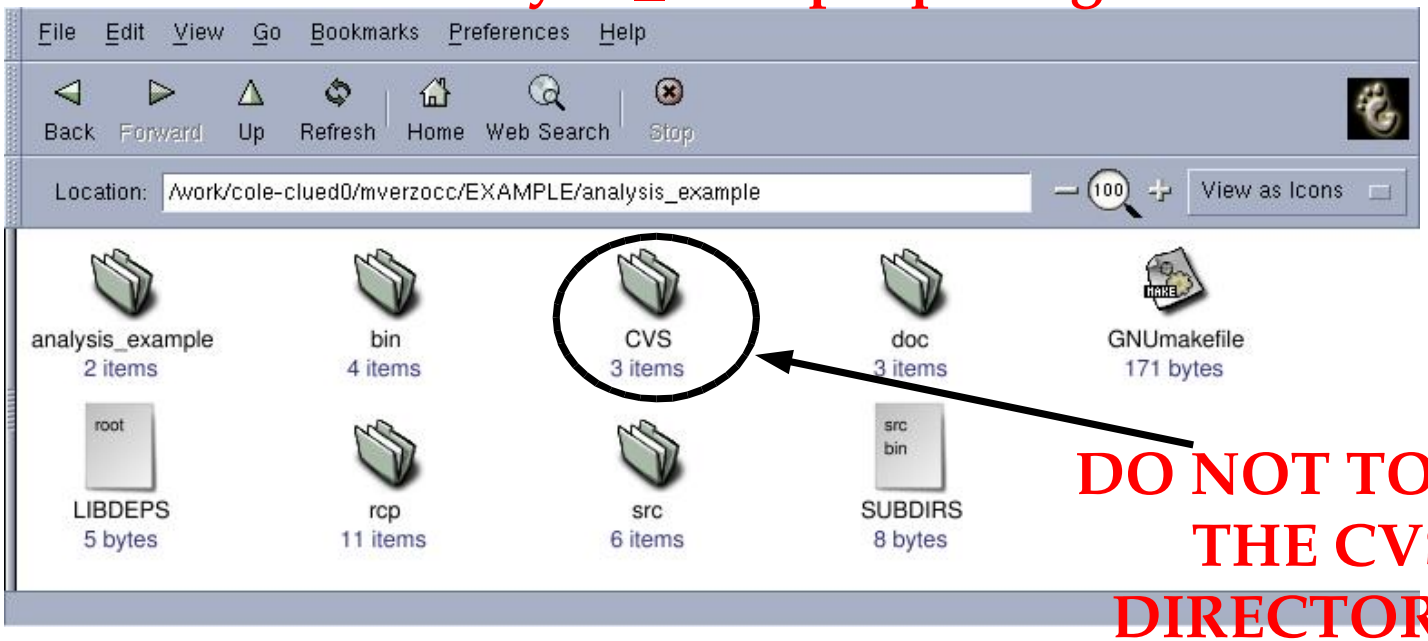
Not yet time to run the program.... First understand the directory structure and what happens in ..../EXAMPLE



After gmake has been done there is much more stuff in there:

- **bin contains the executable(s)**
- **lib contains the libraries**
- **codegen/tmp contain some code/directories created during compilation**
- **rundata contains data possibly needed at runtime**
- **rcpdb contains some RCP control file (created by *d0setwa*)**

Back into the analysis_example package



DO NOT TOUCH THE CVS DIRECTORIES

This is just one possible directory structure for a D0 package: there are several variations. In this scheme:

analysis_example	**header files**
src	**source code, test code**
rcp	**RCP files for this package**
bin	**instructions for building executables**
doc	**documentation**

Other schemes involve: separate test directory

Not so nice: everything in the same directory

Another makefile, usually no need to change it
This one tells gmake to do the standard thing (very few packages really need to do something different)

List of subdirectories in which gmake should do something, in this case src (to build a library and one object file) and bin (to build an executable)

With new building mechanism (t02.25.00 and later) needed only to list the dependencies on external packages (stuff which is not in the /d0dist/dist/releases tree, i.e. ROOT, Pythia, Herwig, ...). For these packages the include statements look like:

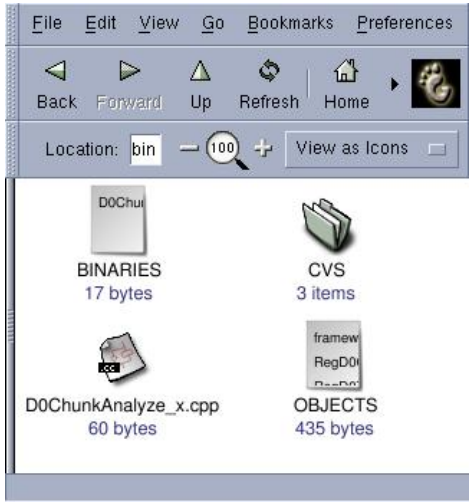
#include "TH1F.h"

whereas for D0 software they are

#include "em_evt/EMparticleChunk.hpp"

Inside the bin directory:

DO NOT TOUCH the CVS subdirectory



3 other files:

BINARIES tells gmake the name of the executable being built (D0ChunkAnalyze_x in this case)

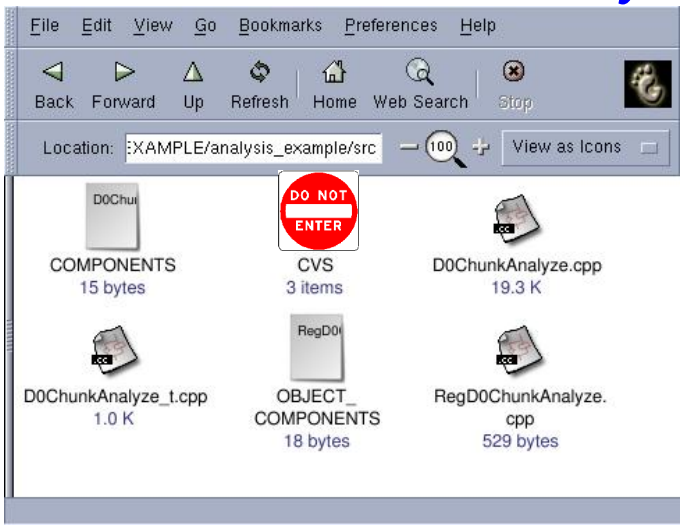
OBJECTS tells gmake which packages should be included in the executable

D0ChunkAnalyze_x.cpp is a dummy file (it contains one comment line and that is all) needed to build the executable. The real main function

(int main(.....) { })

is somewhere else

Come back to OBJECTS in a few slides.... First the src directory



D0ChunkAnalyze.cpp contains the source code for 1 class (third part of this tutorial)

D0ChunkAnalyze_t.cpp contains the source code of a test program (which actually really doesn't do a test)

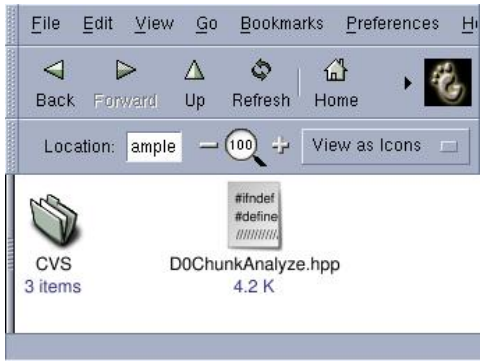
COMPONENTS tells the build system which object files should be put into the libanalysis_example.a library (in lib/Linux2.4-KCC_4_0-maxopt)

RegD0ChunkAnalyze_x.cpp is another (not) simple source file..

OBJECT_COMPONENTS tells the build system which object files should be put directly into the lib/Linux2.4-KCC_4_0-maxopt directory

An object file is what you obtain compiling a source file (D0ChunkAnalyze.o)

OBJECTS, RegD0ChunkAnalyze_x.cpp, OBJECT_COMPONENTS , two more slides.....

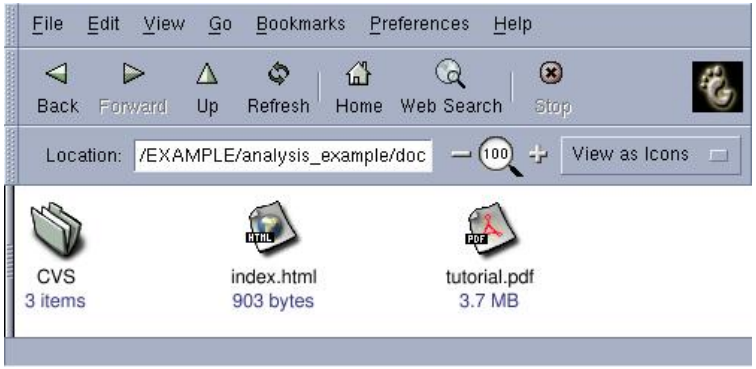


The analysis_example directory contains the header file for the D0ChunkAnalyze class

Remember: EXAMPLE/include contains a link to analysis_example/analysis_example The header files is included as

#include "analysis_example/D0ChunkAnalyze.hpp"

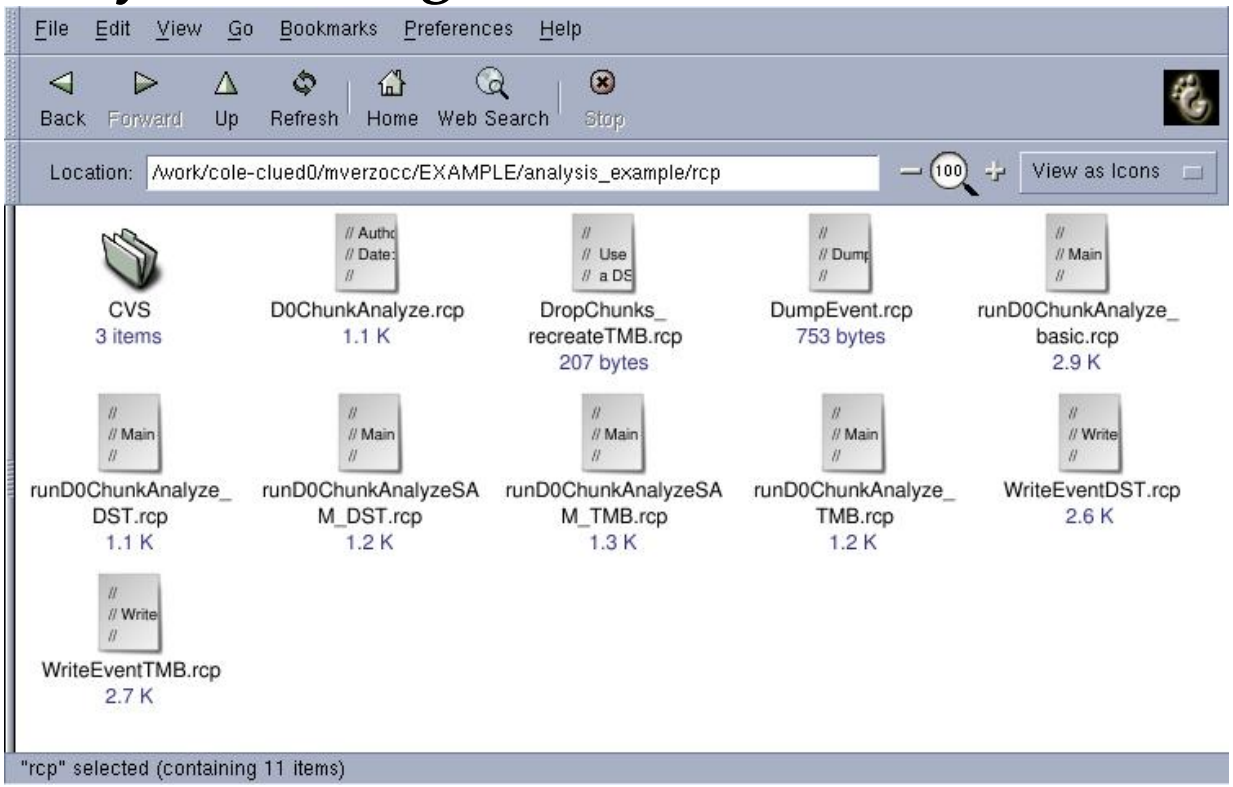
The doc directory is supposed to contain documentation. In this case it contains these transparencies as PDF file(s) (well an earlier version of it .....) plus one HTML file.



This tutorial is available on WWW as

http://www-d0.fnal.gov/d0dist/dist/packages/analysis_example/devel/doc/

The rcp directory contains RCP parameters for this package
we can delay discussing this for another 15 minutes



Now we have to discuss those OBJECTS,
OBJECT_COMPONENTS, RegD0ChunkAnalyze.cpp files.
Then the RCP parameters

The D0 software framework or how to glue together those 1000 packages in different ways and obtain something which works

The problem:

- Read data (I/O, decoding, calibration,.....)
- Reconstruct tracks (Hit finding, track fitting)
- Find vertices (Select tracks, vertex finding, vertex fitting)

- How do we ensure that the appropriate D0 packages are executed in the right order ?
- What if somebody wants to use the same tracks, but a different algorithm to find vertices ?

- How do all those packages share data ?

- How do we keep track of who has done what ?

More problems:

- **Reuse code** (L3, reconstruction, online monitoring, simulation, analysis)
- **Build a flexible system** (turn on/off parts of the code)

Solution

- **D0 software framework** (set of rules to which each package must adhere, so that all packages can work together)
- **RCP parameters** (run control parameters: configure each package and the way packages are executed inside the framework)
- **D0 event model** (how data are shared between different packages)

What is the framework ?

The framework is an object-oriented system that provides the user with a common interface and methodology for doing event analysis. Through a set of classes, the framework allows all users to generate application program modules with a common way in which they receive event data and communicate results. The framework defines the way in which event data flows through user application programs. It provides a set of interface points, which can be utilized to register hooks or functions in an application program. The purpose of the framework is to pass event data from user package interface point to user package interface point, in the order and configuration specified by the user at run time.

It took me more than 1 year to understand these 5 sentences....

The way I understand it:

the framework is a freight train

- **You can put your stuff on a container**
- **You fill a form and ask your container to go from point A to point B, stopping at points C,D,E along the route**
- **At those points you may charge/discharge (i.e. do something) stuff from your container**
- **You may tell the driver not to stop at point D, you've got nothing to do there**

- For this to work your container must conform to the standard (have the right size, have a door which opens on the right side of the tracks) and .....
- You must fill the right papers, telling the railway company where you want to stop and what you want to do.
- Somebody else may get on the train
- They just have to fill the proper forms, have their container. They may ask to do different stops, but as long as everybody is happy to go from A to B, they can still share the ride.
- You also need somebody to drive the train and decide where A and B are, and who is allowed to ride the train (the railway company)

Unfortunately the D0 freight train looks too often like the one in the movie. We've got plenty of bad guys.....

- To complete the analogy: you may want some stuff to be shared among your container and the container belonging to somebody else.
- Usually you put this stuff on pellets. They have a standard format and can be loaded/unloaded with the standard cranes...
- And you need a place where to store all these pellets before you put them on the next train....

- the railway company, the train, the tracks, the stations are the D0 software framework
- the containers are your packages (they do store informations, they do perform actions on this information)
- the forms to be filled to drive the train and to perform certain actions in each container at each stop are the RCP parameters
- the D0 event model is represented by the pellets used to share information
- and the build system allows you to build many trains.....

Thes trains travel a rather stupid railway system.....

At this station tell each container that they can do an action of type A (they may choose to do nothing)

Same thing, action of type B, C, D, E

- At each station you may pass information from one container in the train to the following ones, not viceversa
- But at the next station all the information is available

Building the train allows you to do different things:

have a train which does L3 filtering (here the input file is the DAQ), a train which reconstructs events, a train for analysis.....

Step 1: building the train

- **each train has a name (executable)**
  - **specified by analysis_example/bin/BINARIES**
- **each train needs a steam engine (main program)**
  - **analysis_example/bin/D0ChunkAnalyze_x.cpp (I said a steam engine, it also has a diesel engine.... you just provide the steam one).**
- **tell which wagons/containers should be included**
  - **analysis_example/bin/OBJECTS**

When you compile your code, the build mechanism will ensure that these objects are available inside your train (executable) as well as all the libraries which are needed by these objects....

Mistery #1 solved: we know the purpose of OBJECTS

```
framework
RegD0ChunkAnalyze
RegD0TriggerFilter
RegisterGeometryManagement
RegisterRunConfigPkg
RegistryMagConfigurator
DropChunks
DumpEvent
ReadEvent
WriteEvent
RegSAMManager
LoadDSPACK
LoadEVPACK
LoadCORBA
RegEventTMB
RegThumbNailPkg
RegUnpThumbNailPkg
RegLinkedPhysObjRecc
RegTMBTreePkg
RegTMBTRefsPkg
RegTMBCorePkg
RegTMBTreeMCPkg
RegNtupleMgr
OBJECTS lines 1-32/32 (END)
```

This is analysis_example/bin/OBJECTS

The train infrastructure (framework)
Analysis !!!!!
Filter on triggers ?
Run configuration
Input/output
Talk to SAM
Thumbnail packing/unpacking
ROOT tree
ROOT tuple/histograms

This is the stuff available inside the D0ChunkAnalyze_x executable

And you also know what RegD0ChunkAnalyze.cpp is used for: register your package (a C++ class) with the framework

```
//////////////////////////////////////////////////////////////
// File: RegD0ChunkAnalyze.cpp
//
// Purpose:  register D0 chunk analysis package
//
// Created:    24-JUL-2002  Marco Verzocchi
// History:
//////////////////////////////////////////////////////////////

// Dependencies (#includes)
#include "framework/Registry.hpp"

using namespace fwk;

//////////////////////////////////////////////////////////////
namespace D0example {

FWK_REGISTRY_DECL(D0ChunkAnalyze)

}
RegD0ChunkAnalyze.cpp lines 1-20/20 (END)
```

This is done in a tricky way: the framework doesn't need to know anything about your class, except its name and its address in memory (so that it can pass control to it at each train station).
This also means that the framework can accommodate new classes (wagons) without being changed.

Now we have a train, we run it, decide at which station we're going to stop, and what we'll do at each station.....

Step 2: running the train

- **there is an engineer running the train (the line framework in OBJECTS), it's a wagon itself, it needs instructions....**
  - **read the framework RCP file**

    $> D0ChunkAnalyze_x -rcp runD0ChunkAnalyze_TMB.rcp .....

Each time you run a framework executable you need to pass a file with the -rcp option. This file contains:

- **the stop at which the engineer talks to the railway company**
- **a list of all the possible train stops**
- **a list of all the real train stops**
- **the list of wagons which should really be attached to the train**
- **a list of instructions for each wagon (each wagon is somewhat intelligent, it does something at each train stop)**

Anatomy of a framework RCP: runD0ChunkAnalyze_TMB.rcp

```
//
// Main RCP to run D0ChunkAnalyze on a TMB file.
//

string InterfaceName = "process"
string Interfaces = "generator decide builder filter modify analyze process dump jobSummary runEnd runInit fileOpen fileClose"
string Flow = "generator decide builder filter modify process analyze dump"
```

string InterfaceName = "process"

this is the station at which the engineer is told to stop the train and check with the railway company what to do next

in 99% of the cases it is always "process" (which tells that the program is told to process events)

this allows the program to take a shortcut (not to stop at the following statios for this event) if somebody (one of the passengers of the train) has pulled the alarm and decided that the current event was not good (the filter capability)

Anatomy of a framework RCP: runD0ChunkAnalyze_TMB.rcp

```
//
// Main RCP to run D0ChunkAnalyze on a TMB file.
//

string InterfaceName = "process"
string Interfaces = "generator decide builder filter modify analyze process dump jobSummary runEnd runInit fileOpen fileClose"
string Flow = "generator decide builder filter modify process analyze dump"
```

string Interfaces = "....."

this is the list of all the stations at which the train may decide to stop

at each station each wagon (your C++ class) may perform some action: if your C++ needs to do something at a certain point of the program, it must implement a C++ method (called interface or hook)

if the interface is implemented the framework calls that C++ method for that class and passes to it all the pellets loaded in memory (the event) for processing

Does this mean that when you write your class you should implement all those methods ?

- **No, your class will be modeled after a base class (C++ inheritance) and if you don't implement a method, the corresponding method of the base class will be used**

Why isn't there something like jobStart ?

- **Each container is put on the train automatically as soon as it is requested (in 2 slides), that is implicit in the framework**

What is the meaning of all those train stops ?

- **To be explained in a few slides**

Anatomy of a framework RCP: runD0ChunkAnalyze_TMB.rcp

```
//
// Main RCP to run D0ChunkAnalyze on a TMB file.
//

string InterfaceName = "process"
string Interfaces = "generator decide builder filter modify analyze process dump jobSummary runEnd runInit fileOpen fileClose"
string Flow = "generator decide builder filter modify process analyze dump"

// Control the packages which are executed

str

//
//
//
//

RCP
RCP
RCP
RCP
RCP
RCP

//

int
run
```

string Flow = "....."

this is the order of the train stops in the loop over each event

the order in which the different interfaces of a single class are called one after the other for each event

So far, we've dealt with the train route and train stops, still nothing for the wagons.....

Anatomy of a framework RCP: runD0ChunkAnalyze_TMB.rcp

```
//
// Main RCP to run D0ChunkAnalyze on a TMB file.
//

string InterfaceName = "process"
string Interfaces = "generator decide builder filter modify analyze process dump jobSummary runEnd runInit fileOpen fileClose"
string Flow = "generator decide builder filter modify process analyze dump"

// Control the packages which are executed

string Packages = "geometry read config unpack trigsel analyze"

// Basic part: D0 software initialisation (handling of detector configuration)
//             data I/O
//             unpack the thumbnail
//             select and analyze events

RCP geometry  = <geometry_management geometry_management>  // Configure detector geometry
RCP read      = <d0reco D0recoReadEvent>                   // Read event
```

string Packages = "..... analyze"

this is the list of the wagons, or if you want of the packages which are used. Actually these are just aliases

the order matters: remember that at each station you can pass information from one wagon to the following ones, not viceversa ....

Anatomy of a framework RCP: runD0ChunkAnalyze_TMB.rcp

RCP analyze = <analysis_example D0ChunkAnalyze>

this tells the system to go inside the analysis_example package and pick up from the file D0ChunkAnalyze.rcp instructions for the wagon which is going to perform the analyze action whenever the framework calls one of its methods

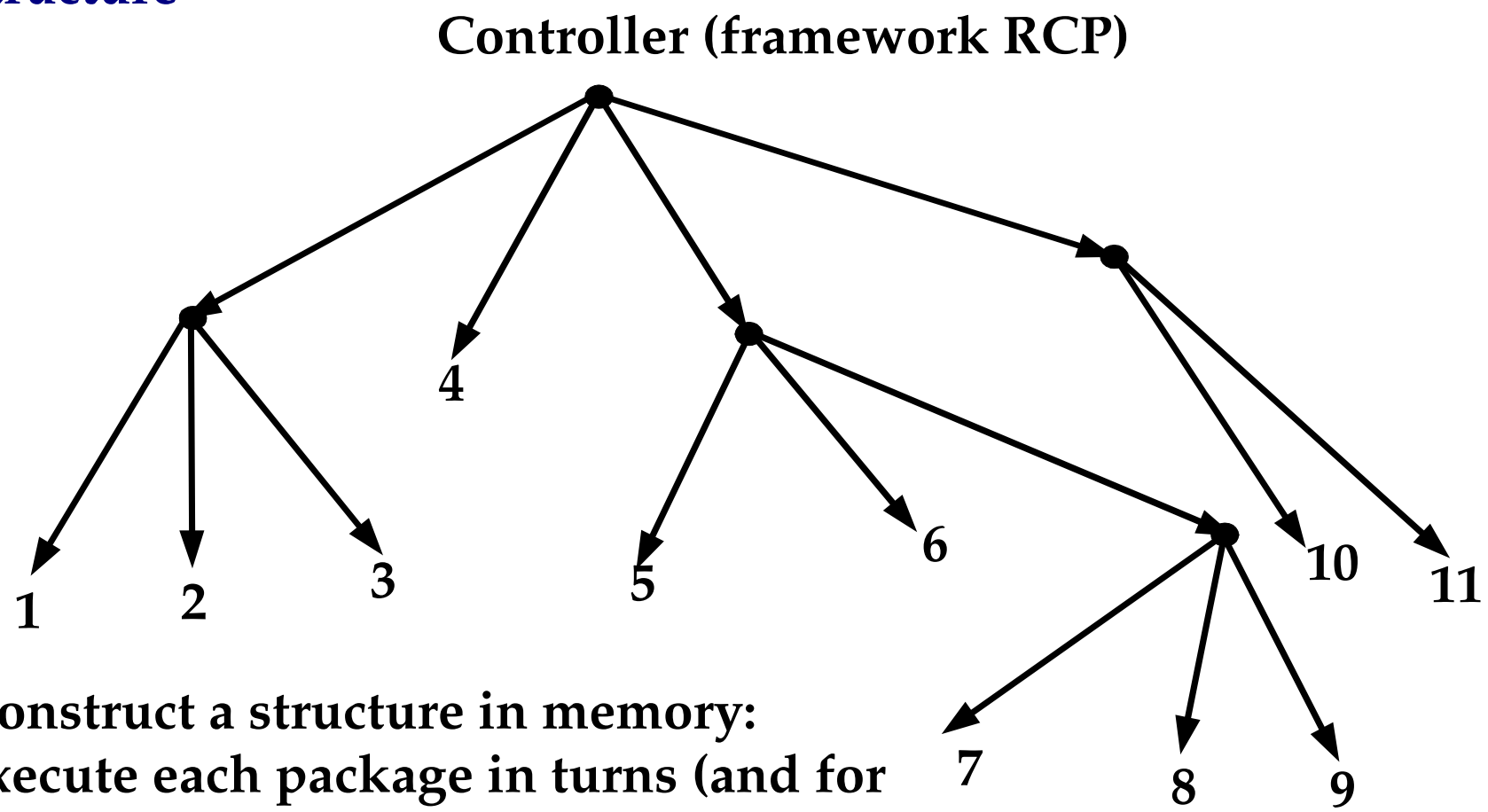
```
RCP unpack    = <thumbnail UnpThumbNail>                // Unpack the thumbnail, recreate D0 physics object chunks
RCP trigsel   = <analysis_utilities D0TriggerFilter>    // Filter events based on triggers
RCP analyze   = <analysis_example D0ChunkAnalyze>       // Run an event selection and fill histograms

// See the runD0ChunkAnalyze_basic.rcp for additional possibilities.

int DumpPeriod = 0
runD0ChunkAnalyze_TMB.rcp lines 1-27/27 (END)
```

And similarly for the other 5 packages: geometry, read, config, unpack, trigsel

The name used in string Packages = " ..... " must be repeated in the RCP ..... = < ..... >

Use something meaningful, not aaa, bbb, ccc, Sally, Joe, ...

Each of these additional RCP files may contain a similar structure



Construct a structure in memory: execute each package in turns (and for each package call the different interfaces)

Establish the program flow

End of part 1
5 minutes break
(not more)