AliRoot Primer

Editor P.Hristov

1 Introduction

1.1 About this primer

The aim of this primer is to give some basic information about the ALICE offline framework (AliRoot) from users perspective. We explain in details the installation procedure, and give examples of some typical use cases: detector description, event generation, particle transport, generation of summable digits, event merging, reconstruction, particle identification, and generation of event summary data. The primer also includes some examples of analysis, and short description of the existing analysis classes in AliRoot. An updated version of the document can be downloaded from `http://aliweb.cern.ch/offline`.

For the reader interested by the AliRoot architecture and by the performance studies done so far, a good starting point is Chapter 4 of the ALICE Physics Performance Report[**?**]. Another important document is the ALICE Computing Technical Design Report[**?**]. Some information contained there has been included in the present document, but most of the details have been omitted.

AliRoot uses the ROOT system as a foundation on which the framework for simulation, reconstruction and analysis is built.

Except for large existing libraries, such as Pythia6 and HIJING, and some remaining legacy code, this framework is based on the Object Oriented programming paradigm, and it is written in C++.

The following packages are needed to install the fully operational software distribution: ROOT, available from `http://root.cern.ch` or using the ROOT CVS repository
`:pserver:cvs@root.cern.ch:/user/cvs`,
and AliRoot from the ALICE offline CVS repository
`:pserver:cvs@alisoft.cern.ch:/soft/cvsroot`. One also has to download one or more particle transport packages. GEANT 3 is available from the ROOT CVS repository. FLUKA library can be obtained from
`http://www.fluka.org`, and GEANT 4 distribution – from
`http://cern.ch/geant4`.

1.2 AliRoot framework

In HEP, a framework is a set of software tools that enables data processing. For example the old CERN Program Library was a toolkit to build a framework. PAW was the first example of integration of tools into a coherent ensemble specifically dedicated to data analysis. The role of the framework is shown in Fig. **??**.

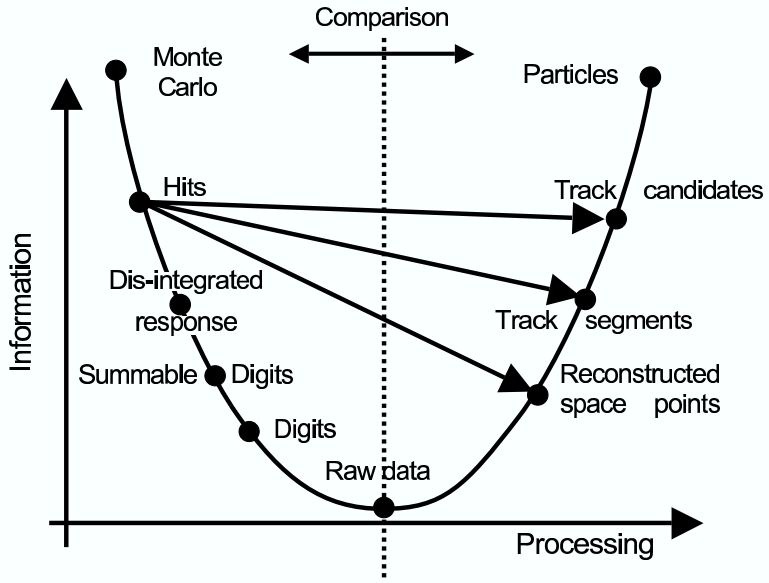

Figure 1: Data processing framework.

The primary interactions are simulated via event generators, and the resulting kinematic tree is then used in the transport package. An event generator produces set of "particles" with their momenta. The set of particles, where one maintains the production history (in form of mother-daughter relationship and production vertex) forms the kinematic tree. More details can be found in the ROOT documentation of class `TParticle`. The transport package transports the particles through the set of detectors, and produces **hits**, which in ALICE terminology means energy deposition at given point. The hits contain also information ("track labels") about the particles that have generated them. There one main exception, namely the calorimeter (PHOS and EMCAL) hits, where a hit is the energy deposition in the whole detecting element volume. In some detectors the energy of the hit is used only for comparison with a given threshold, for example in TOF and ITS pixel layers.

At the next step the detector response is taken into account, and the hits are transformed into **digits**. As it was explained above, the hits are closely related to the tracks which generated them. The transition from hits/tracks to digits/detectors is marked on the picture as "disintegrated response", the tracks are "disintegrated" and only the labels carry the Monte Carlo information. There are two types of digits: **summable digits**, where one uses low thresholds and the result is additive, and **digits**, where the real thresholds are used, and result is similar to what one would get in the real data taking. In some sense the **summable digits** are precursors of the **digits**. The noise simulation is activated when **digits** are produced. There are two differences between the **digits** and the **raw** data format produced by the detector: firstly, the information about the Monte Carlo particle generating the digit is kept, and secondly, the raw data are stored in binary format as "payload" in a ROOT structure, while the digits are stored in ROOT classes. Two conversion chains are provided in AliRoot: **hits** $\rightarrow$ **summable digits** $\rightarrow$ **digits**, and **hits** $\rightarrow$ **digits**. The summable digits are used for the so called "event merging", where a signal event is embedded in a signal-free underlying event. This technique is widely used in heavy-ion physics and allows to reuse the underlying events with substantial economy of computing resources. Optionally it is possible to perform the conversion **digits** $\rightarrow$ **raw data**, which is used to estimate the expected data size, to evaluate the high level trigger algorithms, and to carry on the so called computing data challenges. The reconstruction and the HLT algorithms can work both with **digits** or with **raw data**.

After the creation of digits, the reconstruction and analysis chain can be activated to evaluate the software and the detector performance, and to study some particular signatures. The reconstruction takes as input digits or raw data, real or simulated. The user can intervene into the cycle provided by the framework to replace any part of it with his own code or implement his own analysis of the data. I/O and user interfaces are part of the framework, as are data visualization and analysis tools and all procedures that are considered of general enough interest to be introduced into the framework. The scope of the framework evolves with time as the needs and understanding of the physics community evolves.

The basic principles that have guided the design of the AliRoot framework are re-usability and modularity. There are almost as many definitions of these concepts as there are programmers. However, for our purpose, we adopt an operative heuristic definition that expresses our objective to minimize the amount of unused or rewritten code and maximize the participation of the physicists in the development of the code.

Modularity allows replacement of parts of our system with minimal or no impact on the rest. Not every part of our system is expected to be replaced. Therefore we are aiming at modularity targeted to those elements that we intend to change. For example, we require the ability to change the event generator or the transport

Monte Carlo without affecting the user code. There are elements that we do not plan to interchange, but rather to evolve in collaboration with their authors such as the ROOT I/O subsystem or the ROOT User Interface (UI), and therefore no effort is made to make our framework modular with respect to these. Whenever an element has to be modular in the sense above, we define an abstract interface to it. The codes from the different detectors are independent so that different detector groups can work concurrently on the system while minimizing the interference. We understand and accept the risk that at some point the need may arise to make modular a component that was not designed to be. For these cases, we have elaborated a development strategy that can handle design changes in production code.

Re-usability is the protection of the investment made by the programming physicists of ALICE. The code embodies a large scientific knowledge and experience and is thus a precious resource. We preserve this investment by designing a modular system in the sense above and by making sure that we maintain the maximum amount of backward compatibility while evolving our system. This naturally generates requirements on the underlying framework prompting developments such as the introduction of automatic schema evolution in ROOT.

2 Installation and development tools

2.1 Platforms and compilers

The main development and production platform is Linux on Intel 32 bits processors. Till recently CERN has supported RedHat[**?**] version 7.3, but the code works also with the more recent version 8.0, 9.0, Fedora Core[**?**] 1 – 3, etc. The official Linux[**?**] version at CERN now is Scientific Linux SLC[**?**] 3.0.6. The main compiler on Linux is gcc[**?**]: the recommended version is gcc 3.2 or more recent one, because the older releases (2.91.66, 2.95.2, 2.96) have problems in the FORTRAN optimization. If you use such an old gcc compiler, it is necessary to compile without optimization for the FORTRAN code. As an option you can use Intel icc[**?**] compiler, which is supported as well. You can download it from `http://www.intel.com` and use it free of charge for non-commercial projects. Intel also provides free of charge the VTune[**?**] profiling tool which is really one of the best available so far.

AliRoot is supported on Intel 64 bit processors (Itanium[**?**]) running Linux. Both the gcc and Intel icc compilers can be used.

On 64 bit AMD[**?**] processors such as Opteron AliRoot runs successfully with the gcc compiler.

The software is also regularly compiled and run on other Unix platforms. On Sun (SunOS 5.8) we recommend the CC compiler Sun WorkShop 6 update 1 C++

5.2. The WorkShop integrates nice debugging and profiling facilities which are very useful for code development.

On Compaq alpha server (Digital Unix V4.0) the default compiler is cxx (Compaq C++ V6.2-024 for Digital UNIX V4.0F). Alpha provides also its profiling tool pixie, which works well with shared libraries.

Recently AliRoot was ported to MacOS (Darwin). This OS is very sensitive to the circular dependences in the shared libraries, which makes it very useful as test platform.

Some additional notes concern the new version of the gcc compiler (4.0.0 and later). The gcc developers has decided to abandon the old FORTRAN front end (g77) and to use instead gfortran, which is relatively new product with some known instabilities. For the moment the possibilities to use gcc 4.0.0 – 4.0.2 are being evaluated, but the recommended version is still within the 3.2 – 3.3 release series.

2.2 Essential CVS information

CVS[**?**] stands for Concurrent Version System. It permits to a group of people to work simultaneously on groups of files (for instance program sources). It also records the history of files, which allows back tracking and file versioning. The official CVS Web page is `http://www.cvshome.org/`. CVS has a host of features, among them the most important are:

- CVS facilitates parallel and concurrent code development;
- it provides easy support and simple access;
- it has possibility to establish group permissions (for example only detector experts and CVS administrators can commit code to given detector module).

CVS has rich set of commands, the most important are described below. There exist several tools for visualization, logging and control which work with CVS. More information is available in the CVS documentation and manual[**?**].

Usually the development process with CVS has the following features:

- all developers work on their own copy of the project (in one of their directories)
- they often have to synchronize with a global repository both to update with modifications from other people and to commit their own changes.

Here below we give an example of a typical CVS session

```
# Login to the repository. The password is stored in ~/.cvspass
# If no cvs logout is done, the password remains there and
# one can access the repository without new login
% cvs -d :pserver:hristov@alisoft.cern.ch:/soft/cvsroot login
(Logging in to hristov@alisoft.cern.ch)
CVS password:
xxxxxxxx

# Check-Out a local version of the TPC module
% cvs -d :pserver:hristov@alisoft.cern.ch:/soft/cvsroot checkout TPC
cvs server: Updating TPC
U TPC/.rootrc
U TPC/AliTPC.cxx
U TPC/AliTPC.h
...

# edit file AliTPC.h
# compile and test modifications

# Commit your changes to the repository with an appropriate comment
% cvs commit -m "add include file xxx.h" AliTPC.h
Checking in AliTPC.h;
/soft/cvsroot/AliRoot/TPC/AliTPC.h,v <-- AliTPC.h
new revision: 1.9; previous revision:1.8
done
```

Instead of specifying the repository and user name by -d option, one can export the environment variable CVSROOT, for example

```
% export CVSROOT=:pserver:hristov@alisoft.cern.ch:/soft/cvsroot
```

Once the local version has been checked out, inside the directory tree the CVSROOT is not needed anymore. The name of the actual repository can be found in CVS/Root file. This name can be redefined again using the -d option.

In case somebody else has committed some changes in AliTPC.h file, the developer have to update the local version merging his own changes before committing them:

```
% cvs commit -m "add include file xxx.h" AliTPC.h
cvs server: Up-to-date check failed for ‘AliTPC.h’
```

```
cvs [server aborted]: correct above errors first!

% cvs update
cvs server: Updating .
RCS file: /soft/cvsroot/AliRoot/TPC/AliTPC.h,v
retrieving revision 1.9
retrieving revision 1.10
Merging differences between 1.9 and 1.10 into AliTPC.h

M AliTPC.h
# edit, compile and test modifications

% cvs commit -m "add include file xxx.h" AliTPC.h
Checking in AliTPC.h;
/soft/cvsroot/AliRoot/TPC/AliTPC.h,v <-- AliTPC.h
new revision: 1.11; previous revision: 1.10
done
```

Important note: CVS performs a purely mechanical merging, and it is the developer's to verify the result of this operation. It is especially true in case of conflicts, when the CVS tool is not able to merge the local and remote modifications consistently.

2.3 Main CVS commands

In the following examples we suppose that the CVSROOT environment variable is set, as it was shown above. In case a local version has been already checked out, the CVS repository is defined automatically inside the directory tree.

- **login** stores password in .cvspass. It is enough to login once to the repository.
- **checkout** retrieves the source files

  ```
  % cvs -q -z9 co -r v4-01-Rev-08 AliRoot
  ```

- **update** retrieves modifications from the repository and merges them with the local ones. The -q option reduces the verbose output, and the -z9 sets the compression level during the data transfer.

```
% cvs -q -z9 update -AdP STEER
```

- **diff** shows differences between the local and repository versions

```
% cvs -q -z9 diff STEER
```

- **add** adds files or directories to the repository. The actual transfer is done when the commit command is invoked.

```
% cvs -q -z9 add AliTPCseed.*
```

- **remove** removes old files or directories from the repository. The -f option forces the removal of the local files.

```
% cvs remove -f CASTOR
```

- **commit** checks in the local modifications to the repository and increments the version.

```
% cvs -q -z9 commit -m ‘‘Coding convention’’ STEER
```

- **tag** creates new tags and/or branches (with -b option).

```
% cvs tag -b v4-01-Rev-09 .
```

- **status** returns the actual status of a file: revision, sticky tag, dates, options, and local modifications.

```
% cvs status Makefile
```

- **logout** removes the stored password. It is not really necessary unless the user really wants to remove the password from that account.

2.4 Environment variables

Before the installation of AliRoot the user has to set some environment variables. In the following examples the user is working on Linux and the default shell is bash. It is enough to add to the .bash_profile file few lines as shown below:

```
# ROOT
export ROOTSYS=/home/mydir/root
export PATH=$PATH\:$ROOTSYS/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH\:$ROOTSYS/lib

# AliRoot
export ALICE=/home/mydir/alice
export ALICE_ROOT=$ALICE/AliRoot
export ALICE_TARGET=‘root-config --arch‘
export PATH=$PATH\:$ALICE_ROOT/bin/tgt_${ALICE_TARGET}
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH\:$ALICE_ROOT/lib/tgt_${ALICE_TARGET}

# GEANT 3
export PLATFORM=‘root-config --arch‘
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH\:$ALICE/geant3/lib/tgt_${ALICE_TARGET}
```

where “/home/mydir” has to be replaced with the actual directory path. The meaning of the environment variables is the following:

ROOTSYS – the place where the ROOT package is located;

ALICE – top directory for all the software packages used in ALICE;

ALICE_ROOT – the place where the AliRoot package is located, usually as subdirectory of ALICE;

ALICE_TARGET – specific platform name. Up to release v4-01-Release this variable was set to the result of “uname” command. Starting from AliRoot v4-02-05 the ROOT naming schema was adopted, and the user has to use “root-config –arch” command.

PLATFORM – the same as ALICE_TARGET for the GEANT 3 package. Until GEANT 3 v1-0 the user had to use ‘uname‘ to specify the platform. From version v1-0 on the ROOT platform is used instead (“root-config –arch”).

2.5 Software packages

2.5.1 ROOT

All ALICE offline software is based on ROOT[**?**]. The ROOT framework offers a number of important elements which are exploited in AliRoot:

- a complete data analysis framework including all the PAW features;
- an advanced Graphic User Interface (GUI) toolkit;
- a large set of utility functions, including several commonly used mathematical functions, random number generators, multi-parametric fit and minimization procedures;
- a complete set of object containers;
- integrated I/O with class schema evolution;
- C++ as a scripting language;
- documentation tools.

There is a nice ROOT user's guide which incorporates important and detailed information. For those who are not familiar with ROOT a good staring point is the ROOT Web page at `http://root.cern.ch`. Here the experienced users may find easily the latest version of the class descriptions and search for useful information.

The recommended way to install ROOT is from the CVS sources, as it is shown below:

1. Login to the ROOT CVS repository if you haven't done it yet.

   ```
   % cvs -d :pserver:cvs@root.cern.ch:/user/cvs login
   % CVS password: cvs
   ```

2. Download (check out) the needed ROOT version (v5-08-00 in the example)

   ```
   % cvs -d :pserver:cvs@root.cern.ch:/user/cvs co -r v5-08-02 root
   ```

   A list of matching Root, Geant3 and AliRoot versions can be found at `http://aliweb.cern.ch/offline/releases.html`

3. The code is stored in the directory "root". You have to go there, set the ROOTSYS environment variable (if this is not done in advance),and configure ROOT. The ROOTSYS contains the full path to the ROOT directory.

   Please note that now Root check for libPythia6.* and then searches in the list of found libraries for one which contains the pythia6_common_block_address_ string. In case no library is found or none contains this common block, the compilation of the Pythia6 interface is switched off. The following configuration script overcomes this requirement, since AliRoot uses its own version of Pythia6, and this version is not yet available at the time of Root compilation.

```
% cd root
% export ROOTSYS=‘pwd‘

# The following is a script to provide fake Pythia6 library
# to ROOT which triggers the configuration of the interface
[ -r lib ] || mkdir lib
rm -rf lib/libPythia6.a

cat > /tmp/p1.c <<EOF
void nevercalled() {}
EOF
gcc -c -o /tmp/p1.o /tmp/p1.c

cat > /tmp/p2.c <<EOF
void pythia6_common_block_address_() {}
EOF
gcc -c -o /tmp/p2.o /tmp/p2.c

ar rv lib/libPythia6.a /tmp/p2.o
ranlib lib/libPythia6.a

ALIEN_ROOT=/opt/alien
./configure linux \
      --enable-pythia6 --with-pythia6-libdir=$ROOTSYS/lib \
      --enable-cern --enable-rfio \
      --enable-mathmore --enable-mathcore --enable-roofit \
      --enable-asimage --enable-minuit2 \
      --enable-alien --with-alien-incdir=$ALIEN_ROOT/api/include \
      --with-alien-libdir=$ALIEN_ROOT/api/lib \
      --with-ssl-incdir=$ALIEN_ROOT/include \
      --with-ssl-libdir=$ALIEN_ROOT/lib

rm lib/libPythia6.a
ar rv lib/libPythia6.a /tmp/p1.o
ranlib lib/libPythia6.a
rm /tmp/p1.* /tmp/p2.*
```

4. Now you can compile and test ROOT

```
% make
% cd test
```

```
% make
% export LD_LIBRARY_PATH=$LD_LIBRARY_PATH\:.
% ./stress
% ./stressgeom
% ./stressLinear
% ./stressVector
```

Another possibility to solve the "Pythia6 problem" mentioned above is to compile first Root without Pythia6 support. Then the compilation of AliRoot is started, and it finishes unsuccessfully because the libEGPythia6.so library is missing. However one can use the newly created Pythia6 library from AliRoot to reconfigure ROOT and to create the missing interface (libEGPythia6.so). Here is how this can be done

```
cd $ROOTSYS/lib
ln -s $ALICE_ROOT/lib/tgt_$ALICE_TARGET/libpythia6.so libPythai6.so
cd ..
./configure linux  --enable-pythia6 --with-pythia6-libdir=$ROOTSYS/lib make
```

At this point the user should have a working ROOT version on a Linux (32 bit Pentium processor with gcc compiler). In case of different platform the user has to replace the second argument in the ./configure command. For example if the Itanium version of Root compiled with icc is needed, you have to configure specifying linuxi64icc as platform. The list of platform can be obtained by "./configure –help" command.

2.5.2 GEANT 3

The installation of GEANT 3 is needed since for the moments this is the default particle transport package. GEANT 3 description is available on

`http://wwwasdoc.web.cern.ch/wwwasdoc/geant_html3/geantall.html`

You can download the GEANT 3 distribution from the ROOT CVS repository and compile it in the following way:

```
% cd $ALICE
% cvs -q -z2 -d :pserver:cvs@root.cern.ch:/user/cvs co -r v1-3 geant3
% cd $ALICE/geant3
% export PLATFORM=‘root-config --arch‘
% make
```

Please note that GEANT 3 is downloaded in $ALICE directory. This feature is used in AliRoot to set correctly the include path to TGeant3.h. Another important feature is the PLATFORM environment variable. If it is not set, the makefile sets it to the result of ‘root-config –arch‘.

2.5.3 GEANT 4

To use GEANT 4[**?**], some additional software has to be installed. GEANT 4 needs CLHEP[**?**] package, the user can get the tar file (hereon “tarball”) from `http://proj-clhep.web.cern.ch/proj-clhep/` . Then the installation can be done in the following way:

```
% cd $ALICE
% tar zxvf clhep-2.0.2.2.tgz
% cd CLHEP
% ./configure --prefix=/opt/CLHEP # You can use also ‘pwd‘ or something else
% make
% make check
% make install
```

Another possibility is to use the CLHEP CVS repository:

```
% cvs -d :pserver:anonymous@clhep.cvs.cern.ch:/cvs/CLHEP login # Empty password
% cd $ALICE
% cvs -d :pserver:anonymous@clhep.cvs.cern.ch:/cvs/CLHEP co -r CLHEP_2_0_2_2 CLHEP
% cd CLHEP
% ./bootrstrap
% ./configure --prefix=/opt/CLHEP # You can use also ‘pwd‘ or something else
% make
% make check
% make install
```

Now the following lines should be added to the .bash_profile

```
export CLHEP_BASE_DIR=$ALICE/CLHEP
```

The next step is to install GEANT 4. First of all, you have to download the GEANT 4 tarball from `http://geant4.web.cern.ch/geant4/`

Then the following steps have to be executed:

```
% cd $ALICE
% tar zxvf geant4.8.0.p01.gtar.gz
```

```
% ln -s geant4.8.0.p01 geant4
% cd geant4
% ./Configure -build
# As asnwer choose the default (proposed) value, except of:
# Question: Where is Geant4 installed? /opt/geant4
# Question: Do you want to copy all Geant4 headers in one directory? YES
# Please, specify where CLHEP is installed: /usr/CLHEP
# Question: Do you want to build 'shared' (.so) libraries? YES
# Question: Do you want to use G4VIS_USE_OPENGLX? YES
# Question: Do you want to set G4USE_G3TOG4? YES

# now a long compilation...

% ./Configure -install
% . $ALICE/geant4/env.sh
```

The execution of the env.sh script can be made from the .bash_profile to have the GEANT 4 environment initialized automatically.

2.5.4 FLUKA

The installation of FLUKA[**?**] consists of the following steps:

1. download the latest FLUKA version from `http://www.fluka.org`
2. install

   ```
   % mkdir myFlukaArea
   % mv fluka.tar.gz myFlukaArea/.
   % cd myFlukaArea
   % gunzip fluka.tar.gz
   % tar -xf fluka.tar
   % export FLUPRO=/somePath/myFlukaArea
   ```

3. compile TFluka

   ```
   % cd $ALICE_ROOT
   % make all-TFluka
   ```

4. create a working directory to run aliroot using FLUKA

```
% cd $ALICE_ROOT/TFluka/scripts
% ./runflukageo.sh
```

This script creates the directory tmp and inside all the necessary links for data and configuration files and starts aliroot. For the next run it is not necessary to run the script again. The tmp directory can be kept or renamed. The user should run aliroot from inside this directory.

5. change the Config.C (The configuration file Config.C is explained in details in section “

   When you run

   ```
   gAlice->Init()
   ```

   from the aliroot prompt the program will generate the files containing the cross sections for the electromagnetic processes, with `pemf` extension. Once this file has been created it is not necessary to repeat the procedure for the same geometry. Change in Config.C the line

   ```
   ((TFluka*) gMC)->SetGeneratePemf(kTRUE);
   ```

   to

   ```
   ((TFluka*) gMC)->SetGeneratePemf(kFALSE);
   ```

   In future, this procedure will be automatic. Some important changes are foreseen, namely the creation of pemf files will be automatic.

6. report any problems you encounter to the offline list.

2.5.5 AliRoot

The AliRoot distribution is taken from the CVS repository and then

```
% cd $ALICE
% cvs -q -z2 -d :pserver:cvs@alisoft.cern.ch:/soft/cvsroot co AliRoot
% cd $ALICE_ROOT
% make
```

The AliRoot code (the above example retrieves the HEAD version from CVS) is contained in ALICE_ROOT directory. The ALICE_TARGET is defined automatically in the `.bash_profile` via the call to ‘root-config –arch‘.

2.6 Debugging

While developing code or running some ALICE program, the user may be confronted with the following execution errors:

- floating exceptions: division by zero, sqrt from negative argument, assignment of NaN, etc.
- segmentation violations/faults: attempt to access a memory location that is not allowed to access, or in a way which is not allowed.
- bus error: attempt to access memory that the computer cannot address.

In this case, the user will have to debug the program to determine the source of the problem and fix it. There is several debugging techniques, which are briefly listed below:

- using printf(...) and assert(...)
  - often this is the only easy way to find the origin of the problem;
  - assert(...) aborts the program execution if the argument is FALSE. It is a macro from assert.h, it can be inactivated by compiling with -DNDEBUG.
- using gdb
  - gdb needs compilation with -g option. Sometimes -O -g prevents from exact tracing;
  - One can use it directly (gdb aliroot) or attach it to a process (gdb aliroot 12345).

Below we report the main gdb commands and their descriptions:

- **run** starts the execution of the program;
- **Control-C** stops the execution and switches to the gdb shell;
- **where** prints the program stack. Sometimes the program stack is very long. The user can get the last n frames by specifying n as a parameter to where;
- **print** prints the value of the expression

```
(gdb) print *this
```

- **up** and **down** are used to navigate in the program stack;
- **print** prints a variable or expression;
- **quit** exits the gdb session;
- **break** sets break point;

```
(gdb) break AliLoader.cxx:100
(gdb) break 'AliLoader::AliLoader()'
```

  The automatic completion of the class methods via tab is available in case an opening quote is put in front of the class name.

- **cont** continues the run;
- **watch** sets watchpoint (slow execution). The example below shows how to check each change of fData;

```
(gdb) watch *fData
```

- **list** shows the source code;
- **help** shows the description of commands.

2.7 Profiling

Profiling is used to discover where the program spends most of the time, and to optimize the algorithms. There are several profiling tools available on different platforms:

- Linux: gprof: compilation with -pg option, static libraries oprofile: uses kernel module VTune: instruments shared libraries
- Sun: Sun workshop (Forte agent). It needs compilation with profiling option (-pg)
- Compaq Alpha: pixie profiler. instruments shared libraries for profiling.

On Linux AliRoot can be built with static libraries using the special target "profile"

```
% make profile
# change LD_LIBRARY_PATH to replace lib/tgt_linux with lib/tgt_linuxPROF
# change PATH to  replace bin/tgt_linux with bin/tgt_linuxPROF
% aliroot
root [0] gAlice->Run()
root [1] .q
```

After the end of aliroot session a file called gmon.out will be created. It contains the profiling information which can be investigated using gprof.

```
% gprof `which aliroot` | tee gprof.txt
% more gprof.txt
```

VTune profiling tool

VTune is available from the Intel Web site `http://www.intel.com/software/products/index.htm` It is free for non-commercial use on Linux. It provides possibility for call-graph and sampling profiling. VTuneInstruments shared libraries, and needs only -g option during the compilation. Here is an example of call-graph profiling:

```
# Register an activity
% vtl activity sim -c callgraph -app aliroot,'' -b -q sim.C'' -moi aliroot
% vtl run sim
% vtl show
% vtl view sim::r1 -gui
```

2.8 Detection of run time errors

The Valgrind tool can be used for detection of run time errors on linux. It is available from `http://www.valgrind.org`. Valgrind is equipped with the following set of tools:

- memcheck for memory management problems;
- addrcheck: lightweight memory checker;
- cachegrind: cache profiler;
- massif: heap profiler;
- hellgrind: thread debugger;
- callgrind: extended version of cachegrind.

The most important tool is memcheck. It can detect:

- use of non-initialized memory;
- reading/writing memory after it has been free'd;
- reading/writing off the end of malloc'd blocks;
- reading/writing inappropriate areas on the stack;
- memory leaks – where pointers to malloc'd blocks are lost forever;
- mismatched use of malloc/new/new [] vs free/delete/delete [];
- overlapping source and destination pointers in memcpy() and related functions;
- some misuses of the POSIX pthreads API;

Here is an example of Valgrind usage:

```
% valgrind --tool=addrcheck --error-limit=no aliroot -b -q sim.C
```

ROOT memory checker

The ROOT memory checker provides tests of memory leaks and other problems related to new/delete. It is fast and easy to use. Here is the recipe:

- link aliroot with -lNew. The user has to add 'new' before 'glibs' in the ROOTCLIBS variable of the Makefile;
- add Root.MemCheck: 1 in .rootrc
- run the program: aliroot -b -q sim.C
- run memprobe -e aliroot
- Inspect the files with .info extension that have been generated.

2.9 Useful information LSF and CASTOR

LSF is the batch system at CERN. Every user is allowed to submit jobs to the different queues. Usually the user has to copy some input files (macros, data, executables, libraries) from a local computer or from the mass-storage system to the worker node on lxbatch, than to execute the program, and to store the results on the local computer on in the mass-storage system. The main steps and commands are described below.

In order to have access to the local desktop and to be able to use scp without password, the user has to create pair of SSH keys. Currently lxplus/lxbatch uses RSA1 cryptography. After login into lxplus the following has to be done:

```
% ssh-keygen -t rsa1
# Use empty password
% cp .ssh/identity.pub public/authorized_keys
% ln -s ../public/authorized_keys .ssh/authorized_keys
```

A list of useful LSF commands is given bellow:

- **bqueues** shows the available queues and their status;
- **bsub -q 8nm job.sh** submits the shell script job.sh to the queue 8nm, where the name of the queue indicates the "normalized CPU time" (maximal job duration 8 min of CPU time);
- **bjobs** lists all unfinished jobs of the user;
- **lsrun -m lxbXXXX xterm** returns a xterm running on the batch node lxbXXXX. This permits to inspect the job output and to debug a batch job.

Each batch job store the output in directory LSFJOB_XXXXXX, where XXXXXX is the job id. Since the home directory is on AFS, the user has to redirect the verbose output, otherwise the AFS quota might be exceeded and the jobs will fail.

The CERN mass storage system is CASTOR2[**?**]. Every user has its own CASTOR2 space, for example /castor/cern.ch/user/p/phristov. The commands of CASTOR2 start with prefix "ns" of "rf". Here is very short list of useful commands:

- **nsls /castor/cern.ch/user/p/phristov** lists the CASTOR space of user phristov;
- **rfdir /castor/cern.ch/user/p/phristov** the same as above, but the output is in long format;
- **nsmkdir test** creates a new directory (test) in the CASTOR space of the user;
- **rfcp /castor/cern.ch/user/p/phristov/test/galice.root .** copies the file from CASTOR to the local directory. If the file is on tape, this will trigger the stage-in procedure, which might take some time.
- **rfcp AliESDs.root /castor/cern.ch/p/phristov/test** copies the local file AliESDs.root to CASTOR in the subdirectory test and schedules it for migration to tape.

The user also has to be aware, that the behavior of CASTOR depends on the environment variables RFIO_USE_CASTOR_V2(=YES), STAGE_HOST(=castoralice) and STAGE_SVCCLASS(=default). They are set by default to the values for the group (z2 in case of ALICE).

Below the user can find an example of job, where the simulation and reconstruction are run using the corresponding macros sim.C and rec.C. An example of such macros will be given later.

```
#! /bin/sh
# Take all the C++ macros from the local computer to the working directory
command scp phristov@pcepalice69:/home/phristov/pp/*.C .

# Execute the simulation macro. Redirect the output and error streams
command aliroot -b -q sim.C > sim.log 2>&1

# Execute the reconstruction macro. Redirect the output and error streams
command aliroot -b -q rec.C > rec.log 2>&1

# Create a new CASTOR directory for this job ($LSB_JOBID)
command rfmkdir /castor/cern.ch/user/p/phristov/pp/$LSB_JOBID

# Copy all log files to CASTOR
for a in *.log; do rfcp $a /castor/cern.ch/user/p/phristov/pp/$LSB_JOBID; done
# Copy all ROOT files to CASTOR
for a in *.root; do rfcp $a /castor/cern.ch/user/p/phristov/pp/$LSB_JOBID; done
```

3 Simulation

3.1 Introduction

Heavy-ion collisions produce a very large number of particles in the final state. This is a challenge for the reconstruction and analysis algorithms. The development of these algorithms requires a predictive and precise simulation of the detector response. Model predictions discussed in the first volume of Physics Performance Report for the charged multiplicity at LHC in Pb–Pb collisions vary from 1400 to 8000 particles in the central unit of rapidity. The experiment was designed when the highest nucleon–nucleon center-of-mass energy heavy-ion interactions was at $20\,\mathrm{GeV}$ per nucleon–nucleon pair at CERN SPS, i.e. a factor of about 300 less than the energy at LHC. Recently, the RHIC collider came online. Its top energy of $200\,\mathrm{GeV}$ per nucleon–nucleon pair is still 30 times less than the

LHC energy. The RHIC data seem to suggest that the LHC multiplicity will be on the lower side of the interval. However, the extrapolation is so large that both the hardware and software of ALICE have to be designed for the highest multiplicity. Moreover, as the predictions of different generators of heavy-ion collisions differ substantially at LHC energies, we have to use several of them and compare the results.

The simulation of the processes involved in the transport through the detector of the particles emerging from the interaction is confronted with several problems:

- existing event generators give different answers on parameters such as expected multiplicities, p_T-dependence and rapidity dependence at LHC energies.
- most of the physics signals, like hyperon production, high-p_T phenomena, open charm and beauty, quarkonia etc., are not exactly reproduced by the existing event generators.
- simulation of small cross-sections would demand prohibitively long runs to simulate a number of events that is commensurable with the expected number of detected events in the experiment.
- the existing generators do not provide for event topologies like momentum correlations, azimuthal flow etc.

To allow nevertheless efficient simulations we have adopted a framework that allows for a number of options:

- the simulation framework provides an interface to external generators, like HIJING [**?**] and DPMJET [**?**].
- a parameterized, signal-free, underlying event where the produced multiplicity can be specified as an input parameter is provided.
- rare signals can be generated using the interface to external generators like PYTHIA or simple parameterizations of transverse momentum and rapidity spectra defined in function libraries.
- the framework provides a tool to assemble events from different signal generators (event cocktails).
- the framework provides tools to combine underlying events and signal events at the primary particle level (cocktail) and at the summable digit level (merging).

- afterburners are used to introduce particle correlations in a controlled way. An afterburner is a program which changes the momenta of the particles produced by another generator, and thus modifies as desired the multi-particle momentum distributions.

The implementation of this strategy is described below. The results of different Monte Carlo generators for heavy-ion collisions are described in section **??**.

3.2 Simulation framework

The simulation framework covers the simulation of primary collisions and generation of the emerging particles, the transport of particles through the detector, the simulation of energy depositions (hits) in the detector components, their response in form of so called summable digits, the generation of digits from summable digits with the optional merging of underlying events and the creation of raw data. The `AliSimulation` class provides a simple user interface to the simulation framework. This section focuses on the simulation framework from the (detector) software developers point of view.

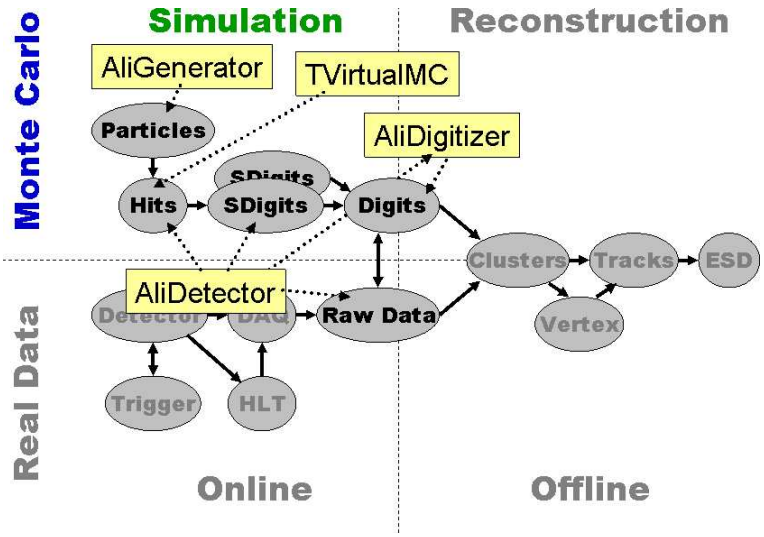

Figure 2: Simulation framework.

Generation of Particles

Different generators can be used to produce particles emerging from the collision. The class `AliGenerator` is the base class defining the virtual interface to the generator programs. The generators are described in more detail in the ALICE PPR Volume 1 and in the next chapter.

Virtual Monte Carlo

The simulation of particles traversing the detector components is performed by a class derived from `TVirtualMC`. The Virtual Monte Carlo also provides an interface to construct the geometry of detectors. The task of the geometry description is done by the geometrical modeler `TGeo`. The concrete implementation of the virtual Monte Carlo application TVirtualMCApplication is AliMC. The Monte Carlos used in ALICE are GEANT 3.21, GEANT 4 and FLUKA. More information can be found on the VMC Web page: `http://root.cern.ch/root/vmc`

As explained above, our strategy was to develop a virtual interface to the detector simulation code. We call the interface to the transport code virtual Monte Carlo. It is implemented via C++ virtual classes and is schematically shown in Fig. **??**. The codes that implement the abstract classes are real C++ programs or wrapper classes that interface to FORTRAN programs.

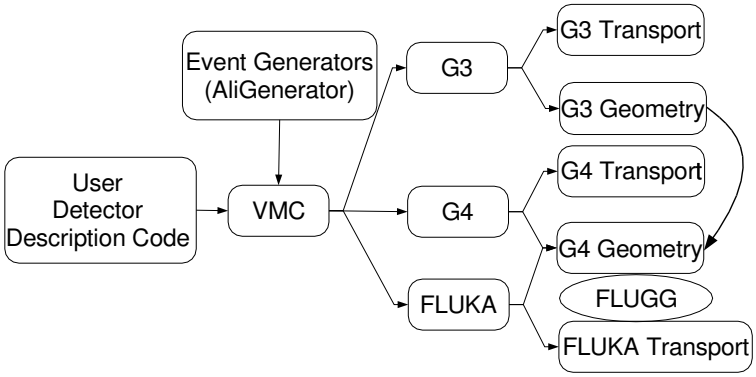

Figure 3: Virtual Monte Carlo

Thanks to the virtual Monte Carlo we have converted all FORTRAN user code developed for GEANT 3 into C++, including the geometry definition and the user scoring routines, `StepManager`. These have been integrated in the detector classes of the AliRoot framework. The output of the simulation is saved directly with ROOT I/O, simplifying the development of the digitization and reconstruction code in C++.

Modules and Detectors

Each module of the ALICE detector is described by a class derived from `AliModule`. Classes for active modules (= detectors) are not derived directly from `AliModule` but from its subclass `AliDetector`. These base classes define the in-

terface to the simulation framework via a set of virtual methods.

Configuration File (Config.C)

The configuration file is a C++ macro that is processed before the simulation starts. It creates and configures the Monte Carlo object, the generator object, the magnetic field map and the detector modules. A detailed description is given below.

Detector Geometry

The virtual Monte Carlo application creates and initializes the geometry of the detector modules by calling the virtual functions `CreateMaterials`, `CreateGeometry`, `Init` and `BuildGeometry`.

Vertexes and Particles

In case the simulated event is intended to be merged with an underlying event, the primary vertex is taken from the file containing the underlying event by using the vertex generator `AliVertexGenFile`. Otherwise the primary vertex is generated according to the generator settings. Then the particles emerging from the collision are generated and put on the stack (an instance of `AliStack`). The transport of particles through the detector is performed by the Monte Carlo object. The decay of particles is usually handled by the external decayer `AliDecayerPythia`.

Hits and Track References

The Monte Carlo simulates the transport of a particle step by step. After each step the virtual method `StepManager` of the module in which the particle currently is located is called. In this step manager method, the hits in the detector are created by calling `AddHit`. Optionally also track references (location and momentum of simulated particles at selected places) can be created by calling `AddTackReference`. `AddHit` has to be implemented by each detector whereas `AddTackReference` is already implemented in AliModule. The container and the branch for the hits – and for the (summable) digits – are managed by the detector class via a set of so-called loaders. The relevant data members and methods are fHits, fDigits, `ResetHits`, `ResetSDigits`, `ResetDigits`,`MakeBranch` and `SetTreeAddress`.

For each detector methods like `PreTrack`, `PostTrack`, `FinishPrimary`, `FinishEvent` and `FinishRun` are called during the simulation when the conditions indicated by the method names are fulfilled.

Summable Digits

Summable digits are created by calling the virtual method Hits2SDigits of a detector. This method loops over all events, creates the summable digits from hits and stores them in the sdigits file(s).

Digitization and Merging

Dedicated classes derived from `AliDigitizer` are used for the conversion of summable digits into digits. Since `AliDigitizer` is a `TTask`, this conversion is done for the current event by the `Exec` method. Inside this method the summable

digits of all input streams have to be added, combined with noise, converted to digital values taking into account possible thresholds and stored in the digits container.

The input streams (more than one in case of merging) as well as the output stream are managed by an object of type `AliRunDigitizer`. The methods GetNinputs, GetInputFolderName and GetOutputFolderName return the relevant information. The run digitizer is accessible inside the digitizer via the protected data member fManager. If the flag fRegionOfInterest is set, only detector parts where summable digits from the signal event are present should be digitized. If Monte Carlo labels are assigned to digits, the stream-dependent offset given by the method `GetMask` is added to the label of the summable digit.

The detector specific digitizer object is created in the virtual method CreateDigitizer of the concrete detector class. The run digitizer object is used to construct the detector digitizer. The `Init` method of each digitizer is called before the loop over the events starts.

A direct conversion from hits directly to digits can be implemented in the method `Hits2Digits` of a detector. The loop over the events is inside the method. Of course merging is not supported in this case.

An example of simulation script that can be used for simulation of proton-proton collisions is provided below:

```
void sim(Int_t nev=100) {
  AliSimulation simulator;

  // Measure the total time spent in the simulation
  TStopwatch timer;
  timer.Start();
  // List of detectors, where both summable digits and digits are provided
  simulator.SetMakeSDigits("TRD TOF PHOS EMCAL RICH MUON ZDC PMD FMD START VZERO");
  // Direct conversion of hits to digits for faster processing (ITS TPC)
  simulator.SetMakeDigitsFromHits("ITS TPC");
  simulator.Run(nev);
  timer.Stop();
  timer.Print();
}
```

The following example shows how one can do event merging

```
void sim(Int_t nev=6) {
  AliSimulation simulator;
```

```
  TStopwatch timer;
  timer.Start();
  // The underlying events are stored in a separate directory.
  // Three signal events will be merged in turn with each
  // underlying event
  simulator.MergeWith("../backgr/galice.root",3);
  simulator.Run(nev);
  timer.Stop();
  timer.Print();
}
```

Raw Data

The digits stored in ROOT containers can be converted into the DATE[**?**] format that will be the 'payload' of the ROOT classes containing the raw data. This is done for the current event in the method `Digits2Raw` of the detector.

The simulation of raw data is managed by the class `AliSimulation`. To create raw data DDL files it loops over all events. For each event it creates a directory, changes to this directory and calls the method `Digits2Raw` of each selected detector. In the Digits2Raw method the DDL files of a detector are created from the digits for the current event.

For the conversion of the DDL files to a DATE file the `AliSimulation` class uses the tool dateStream. To create a raw data file in ROOT format with the DATE output as payload the program alimdc is utilized.

The only part that has to be implemented in each detector is the `Digits2Raw` method of the detectors. In this method one file per DDL has to be created obeying the conventions for file names and DDL IDs. Each file is a binary file with a DDL data header in the beginning. The DDL data header is implemented in the structure `AliRawDataHeader`. The data member fSize should be set to the total size of the DDL raw data including the size of the header. The attribute bit 0 should be set by calling the method `SetAttribute(0)` to indicate that the data in this file is valid. The attribute bit 1 can be set to indicate compressed raw data.

The detector-specific raw data are stored in the DDL files after the DDL data header. The format of this raw data should be as close as possible to the one that will be delivered by the detector. This includes the order in which the channels will be read out.

In order to create RAW data one needs the executable "dateStream" in the actual $PATH. A recent version is always available on AFS the directory /afs/cern.ch/alice/library/local/bin/.

Below we show an example of raw data creation for all the detectors

```
void sim(Int_t nev=1) {
  AliSimulation simulator;
```

```
  TStopwatch timer;
  timer.Start();
  simulator.SetWriteRawData("ALL","raw.root",kFALSE);
  simulator.Run(nev);
  timer.Stop();
  timer.Print();
}
```

3.3 Configuration: example of Config.C

The example below contains as comments the most important information:

```
// Function converting pseudorapidity
// interval to polar angle interval. It is used to set
// the limits in the generator
Float_t EtaToTheta(Float_t arg){
  return (180./TMath::Pi())*2.*atan(exp(-arg));
}

// Switch for holes in front of PHOS. This is a parameter in the
// TRD and TOF geometries
enum PprGeo_t
{
  kHoles, kNoHoles
};

static PprGeo_t geo = kNoHoles; // without holes

void Config()
{
  // Set Random Number seed using the current time
  TDatime dat;
  static UInt_t sseed = dat.Get();
  gRandom->SetSeed(sseed);
  cout<<"Seed for random number generation= "<<gRandom->GetSeed()<<endl;

  // Load GEANT 3 library. It has to be in LD_LIBRARY_PATH
  gSystem->Load("libgeant321");
```

```
// Instantiation of the particle transport package.
// gMC is set internaly
new TGeant3TGeo("C++ Interface to Geant3");

// Create run loader and set some properties
AliRunLoader* rl = AliRunLoader::Open("galice.root",
                         AliConfig::GetDefaultEventFolderName(),
                        "recreate");
if (!rl) Fatal("Config.C","Can not instatiate the Run Loader");
rl->SetCompressionLevel(2);
rl->SetNumberOfEventsPerFile(3);

// Register the run loader in gAlice
gAlice->SetRunLoader(rl);

// Set External decayer
TVirtualMCDecayer *decayer = new AliDecayerPythia();
decayer->SetForceDecay(kAll); // kAll means no specific decay is forced
decayer->Init();

// Register the external decayer in the transport package
gMC->SetExternalDecayer(decayer);

// STEERING parameters FOR ALICE SIMULATION
// Specify event type to be transported through the ALICE setup
// All positions are in cm, angles in degrees, and P and E in GeV
// For the details see the GEANT 3 manual

// Swithch on/off the physics processes
gMC->SetProcess("DCAY",1);
gMC->SetProcess("PAIR",1);
gMC->SetProcess("COMP",1);
gMC->SetProcess("PHOT",1);
gMC->SetProcess("PFIS",0);
gMC->SetProcess("DRAY",0);
gMC->SetProcess("ANNI",1);
gMC->SetProcess("BREM",1);
gMC->SetProcess("MUNU",1);
gMC->SetProcess("CKOV",1);
gMC->SetProcess("HADR",1);
gMC->SetProcess("LOSS",2);
```

```
gMC->SetProcess("MULS",1);
gMC->SetProcess("RAYL",1);

// Set the transport package cuts
Float_t cut = 1.e-3;        // 1MeV cut by default
Float_t tofmax = 1.e10;

gMC->SetCut("CUTGAM", cut);
gMC->SetCut("CUTELE", cut);
gMC->SetCut("CUTNEU", cut);
gMC->SetCut("CUTHAD", cut);
gMC->SetCut("CUTMUO", cut);
gMC->SetCut("BCUTE",  cut);
gMC->SetCut("BCUTM",  cut);
gMC->SetCut("DCUTE",  cut);
gMC->SetCut("DCUTM",  cut);
gMC->SetCut("PPCUTM", cut);
gMC->SetCut("TOFMAX", tofmax);

// Set up the particle generation

// AliGenCocktail permits to combine several different generators
AliGenCocktail *gener = new AliGenCocktail();

// The phi range is always inside 0-360
gener->SetPhiRange(0, 360);

// Set pseudorapidity range from -8 to 8.
Float_t thmin = EtaToTheta(8);   // theta min. <---> eta max
Float_t thmax = EtaToTheta(-8);  // theta max. <---> eta min
gener->SetThetaRange(thmin,thmax);

gener->SetOrigin(0, 0, 0);  // vertex position
gener->SetSigma(0, 0, 5.3);   // Sigma in (X,Y,Z) (cm) on IP position
gener->SetCutVertexZ(1.);     // Truncate at 1 sigma
gener->SetVertexSmear(kPerEvent);

// First cocktail component: 100 ``background'' particles
AliGenHIJINGpara *hijingparam = new AliGenHIJINGpara(100);
hijingparam->SetMomentumRange(0.2, 999);
gener->AddGenerator(hijingparam,"HIJING PARAM",1);
```

```
// Second cocktail component: one gamma in PHOS direction
AliGenBox *genbox = new AliGenBox(1);
genbox->SetMomentumRange(10,11.);
genbox->SetPhiRange(270.5,270.7);
genbox->SetThetaRange(90.5,90.7);
genbox->SetPart(22);
gener->AddGenerator(genbox,"GENBOX GAMMA for PHOS",1);

gener->Init(); // Initialization of the coctail generator

// Field (the last parameter is 1 => L3 0.4 T)
AliMagFMaps* field = new AliMagFMaps("Maps","Maps", 2, 1., 10., 1);
gAlice->SetField(field);

// Make sure the current ROOT directory is in galice.root
rl->CdGAFile();

// ALICE BODY parameters. BODY is always present
AliBODY *BODY = new AliBODY("BODY", "ALICE envelop");

// Start with Magnet since detector layouts may be depending
// on the selected Magnet dimensions
AliMAG *MAG = new AliMAG("MAG", "Magnet");

// Absorber
AliABSO *ABSO = new AliABSOv0("ABSO", "Muon Absorber");

// Dipole magnet
AliDIPO *DIPO = new AliDIPOv2("DIPO", "Dipole version 2");

// Hall
AliHALL *HALL = new AliHALL("HALL", "ALICE Hall");

// Space frame
AliFRAMEv2 *FRAME = new AliFRAMEv2("FRAME", "Space Frame");
if (geo == kHoles) {
  FRAME->SetHoles(1);
} else {
  FRAME->SetHoles(0);
}
```

```
// Shielding
AliSHIL *SHIL = new AliSHILv2("SHIL", "Shielding Version 2");

// Beam pipe
AliPIPE *PIPE = new AliPIPEv0("PIPE", "Beam Pipe");

// ITS parameters
AliITSvPPRasymmFMD *ITS  = new AliITSvPPRasymmFMD("ITS",
                    "ITS PPR detailed version with asymmetric services");
ITS->SetMinorVersion(2);  // don't change it if you're not an ITS developer
ITS->SetReadDet(kTRUE);   // don't change it if you're not an ITS developer
ITS->SetThicknessDet1(200.);   // detector thickness on layer 1
// must be in the range [100,300]
ITS->SetThicknessDet2(200.);   // detector thickness on layer 2
                               // must be in the range [100,300]
ITS->SetThicknessChip1(200.);  // chip thickness on layer 1
                               // must be in the range [150,300]
ITS->SetThicknessChip2(200.);  // chip thickness on layer 2
                               // must be in the range [150,300]
ITS->SetRails(0);        // 1 --> rails in ; 0 --> rails out
ITS->SetCoolingFluid(1);       // 1 --> water ; 0 --> freon

ITS->SetEUCLID(0);  // no output for the EUCLID CAD system

// TPC
AliTPC *TPC = new AliTPCv2("TPC", "Default");

// TOF
AliTOF *TOF = new AliTOFv4T0("TOF", "normal TOF");

// HMPID
AliRICH *RICH = new AliRICHv1("RICH", "normal RICH");

// ZDC
AliZDC *ZDC = new AliZDCv2("ZDC", "normal ZDC");

// TRD parameters
AliTRD *TRD = new AliTRDv1("TRD", "TRD slow simulator");

// Select the gas mixture (0: 97% Xe + 3% isobutane, 1: 90% Xe + 10% CO2)
```

```
  TRD->SetGasMix(1);
  if (geo == kHoles) {
    TRD->SetPHOShole();
    TRD->SetRICHhole();
  }
  // Switch on TR
  AliTRDsim *TRDsim = TRD->CreateTR();

  // FMD
  AliFMD *FMD = new AliFMDv1("FMD", "normal FMD");

  // MUON parameters
  AliMUON *MUON = new AliMUONv1("MUON", "default");
  MUON->AddGeometryBuilder(new AliMUONSt1GeometryBuilder(MUON));
  MUON->AddGeometryBuilder(new AliMUONSt2GeometryBuilder(MUON));
  MUON->AddGeometryBuilder(new AliMUONSlatGeometryBuilder(MUON));
  MUON->AddGeometryBuilder(new AliMUONTriggerGeometryBuilder(MUON));

  // PHOS
  AliPHOS *PHOS = new AliPHOSv1("PHOS", "IHEP");

  // PMD
  AliPMD *PMD = new AliPMDv1("PMD", "normal PMD");

  // START
  AliSTART *START = new AliSTARTv1("START", "START Detector");

  // EMCAL
  AliEMCAL *EMCAL = new AliEMCALv2("EMCAL", "SHISH");

  // VZERO
  AliVZERO *VZERO = new AliVZEROv5("VZERO", "normal VZERO");
}
```

3.4 Event generation

This is from the software chapter in PPR I, it will be modified

The theoretical uncertainty that concerns high-energy heavy-ion collisions at the LHC has several consequences for our simulation strategy. A large part of the physics analysis will be the search for rare signals over an essentially uncorrelated

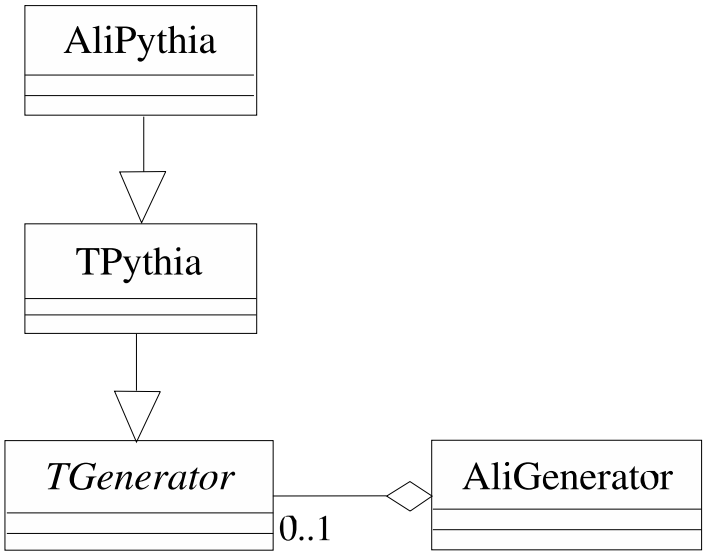

Figure 4: `AliGenerator` is the base class, which has the responsibility to generate the primary particles of an event. Some realizations of this class do not generate the particles themselves but delegate the task to an external generator like PYTHIA through the `TGenerator` interface.

background of emitted particles. To avoid being dependent on a specific model, and to gain in efficiency and flexibility, we use a specially developed parameterization of a signal-free final state. This is already sufficient for a large number of cases as some studies can be done with a single event. This is the case of the track reconstruction efficiency as a function of the initial multiplicity and occupation. This parameterization is based on a pseudo-rapidity (η) parameterization according to HIJING and a transverse momentum (p_T) distribution taken from CDF [**?**] data. To simulate the highest anticipated multiplicities we scale the η-distribution so that 8000 charged particles per event are produced in the range $|\eta| < 0.5$.

To facilitate the usage of different generators we have developed an abstract generator interface called `AliGenerator`, see Fig. **??**. The objective is to provide the user with an easy and coherent way to study a variety of physics signals as well as full set of tools for testing and background studies. This interface allows the study of full events, signal processes, and a mixture of both, i.e. cocktail events.

Several event generators are available via the abstract ROOT class that implements the generic generator interface, `TGenerator`. Through implementations of this abstract base class we wrap FORTRAN Monte Carlo codes like PYTHIA,

HERWIG, and HIJING that are thus accessible from the AliRoot classes. In particular the interface to PYTHIA includes the use of nuclear structure functions of PDFLIB.

In many cases, the expected transverse momentum and rapidity distributions of particles are known. In other cases the effect of variations in these distributions must be investigated. In both situations it is appropriate to use generators that produce primary particles and their decays sampling from parametrized spectra. To meet the different physics requirements in a modular way, the parameterizations are stored in independent function libraries wrapped into classes that can be plugged into the generator. This is schematically illustrated in Fig. **??** where four different generator libraries can be loaded via the abstract generator interface.

It is customary in heavy-ion event generation to superimpose different signals on an event to tune the reconstruction algorithms. This is possible in AliRoot via the so-called cocktail generator (Fig. **??**). This creates events from user-defined particle cocktails by choosing as ingredients a list of particle generators.

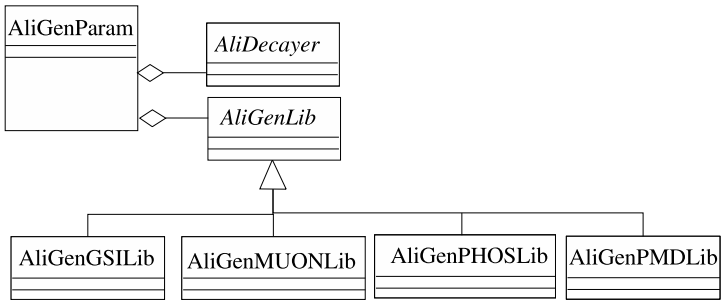

Figure 5: `AliGenParam` is a realization of `AliGenerator` that generates particles using parameterized p_{t} and pseudo-rapidity distributions. Instead of coding a fixed number of parameterizations directly into the class implementations, user defined parametrization libraries (AliGenLib) can be connected at run time allowing for maximum flexibility.

An example of `AliGenParam` usage is presented below:

```
// Example for J/psi  Production from  Parameterisation
// using default library (AliMUONlib)
AliGenParam *gener = new AliGenParam(ntracks, AliGenMUONlib::kUpsilon);
gener->SetMomentumRange(0,999); // Wide cut on the Upsilon momentum
gener->SetPtRange(0,999);       // Wide cut on Pt
gener->SetPhiRange(0. , 360.);  // Full azimutal range
gener->SetYRange(2.5,4);        // In the acceptance of the MUON arm
```

```
gener->SetCutOnChild(1);         // Enable cuts on Upsilon decay products
gener->SetChildThetaRange(2,9); // Theta range for the decay products
gener->SetOrigin(0,0,0);         // Vertex position
gener->SetSigma(0,0,5.3);        // Sigma in (X,Y,Z) (cm) on IP position
gener->SetForceDecay(kDiMuon);  // Upsilon->mu+ mu- decay
gener->SetTrackingFlag(0);       // No particle transport
gener->Init()
```

3.4.1 Pythia6

Pythia is used for simulation of proton-proton interactions and for generation of jets in case of event merging. An example of minimum bias Pythia events is presented below:

```
AliGenPythia *gener = new AliGenPythia(-1);
gener->SetMomentumRange(0,999999);
gener->SetThetaRange(0., 180.);
gener->SetYRange(-12,12);
gener->SetPtRange(0,1000);
gener->SetProcess(kPyMb);         // Min. bias events
gener->SetEnergyCMS(14000.);      // LHC energy
gener->SetOrigin(0, 0, 0);        // Vertex position
gener->SetSigma(0, 0, 5.3);       // Sigma in (X,Y,Z) (cm) on IP position
gener->SetCutVertexZ(1.);         // Truncate at 1 sigma
gener->SetVertexSmear(kPerEvent);// Smear per event
gener->SetTrackingFlag(1);        // Particle transport
gener->Init()
```

3.4.2 HIJING and HIJING parametrization

HIJING (Heavy-Ion Jet Interaction Generator) combines a QCD-inspired model of jet production [**?**] with the Lund model [**?**] for jet fragmentation. Hard or semi-hard parton scatterings with transverse momenta of a few GeV are expected to dominate high-energy heavy-ion collisions. The HIJING model has been developed with special emphasis on the role of mini jets in pp, pA and A–A reactions at collider energies.

Detailed systematic comparisons of HIJING results with a wide range of data demonstrates a qualitative understanding of the interplay between soft string dynamics and hard QCD interactions. In particular, HIJING reproduces many inclusive spectra, two-particle correlations, and the observed flavor and multiplicity dependence of the average transverse momentum.

The Lund FRITIOF [**?**] model and the Dual Parton Model [**?**] (DPM) have guided the formulation of HIJING for soft nucleus–nucleus reactions at intermediate energies, $\sqrt{s_{\rm NN}} \approx 20\,GeV$. The hadronic-collision model has been inspired by the successful implementation of perturbative QCD processes in PYTHIA [**?**]. Binary scattering with Glauber geometry for multiple interactions are used to extrapolate to pA and A–A collisions.

Two important features of HIJING are jet quenching and nuclear shadowing. Jet quenching is the energy loss by partons in nuclear matter. It is responsible for an increase of the particle multiplicity at central rapidities. Jet quenching is modeled by an assumed energy loss by partons traversing dense matter. A simple color configuration is assumed for the multi-jet system and the Lund fragmentation model is used for the hadronisation. HIJING does not simulate secondary interactions.

Shadowing describes the modification of the free nucleon parton density in the nucleus. At the low-momentum fractions, x, observed by collisions at the LHC, shadowing results in a decrease of the multiplicity. Parton shadowing is taken into account using a parameterization of the modification.

Here is an example of event generation with HIJING:

```
AliGenHijing *gener = new AliGenHijing(-1);
gener->SetEnergyCMS(5500.); // center of mass energy
gener->SetReferenceFrame("CMS"); // reference frame
gener->SetProjectile("A", 208, 82); // projectile
gener->SetTarget    ("A", 208, 82); // projectile
gener->KeepFullEvent(); // HIJING will keep the full parent child chain
gener->SetJetQuenching(1); // enable jet quenching
gener->SetShadowing(1); // enable shadowing
gener->SetDecaysOff(1); // neutral pion and heavy particle decays switched off
gener->SetSpectators(0); // Don't track spectators
gener->SetSelectAll(0); // kinematic selection
gener->SetImpactParameterRange(0., 5.); // Impact parameter range (fm)
gener->Init()
```

`AliGenHIJINGparam` [**?**] is an internal AliRoot generator based on parameterized pseudorapidity density and transverse momentum distributions of charged and neutral pions and kaons. The pseudorapidity distribution was obtained from a HIJING simulation of central Pb–Pb collisions and scaled to a charged-particle multiplicity of 8000 in the pseudo rapidity interval $|\eta| < 0.5$. Note that this is about 10% higher than the corresponding value for a rapidity density with an average $\mathrm{d}N/\mathrm{d}y$ of 8000 in the interval $|y| < 0.5$.

The transverse-momentum distribution is parameterized from the measured CDF pion p_T-distribution at $\sqrt{s} = 1.8\,TeV$. The corresponding kaon p_T-distribution

was obtained from the pion distribution by m_T-scaling. See Ref. [**?**] for the details of these parameterizations.

3.4.3 Additional universal generators

Dual parton model

DPMJET is an implementation of the two-component Dual Parton Model (DPM) for the description of interactions involving nuclei based on the Glauber–Gribov approach. DPMJET treats soft and hard scattering processes in an unified way. Soft processes are parameterized according to Regge phenomenology while lowest order perturbative QCD is used to simulate the hard component. Multiple-parton interactions in each individual hadron (nucleon, photon)–nucleon interaction are described by the PHOJET event generator. The fragmentation of parton configurations is treated by the Lund model PYTHIA.

Particle production in the fragmentation regions of the participating nuclei is described by a formation zone suppressed intra-nuclear cascade followed by Monte Carlos of evaporation processes of light nucleons and nuclei, high-energy fission, spectator fragmentation (limited to light-spectator nuclei), and de-excitation of residual nuclei by photon emission.

The most important features of DPMJET-II.5 [**?**] are new diagrams contributing to baryon stopping and a better calculation of Glauber cross-sections. A new striking feature of hadron production in nuclear collisions is the large stopping of the participating nucleons in hadron–nucleus and nucleus–nucleus collisions compared to hadron–hadron collisions [**?**, **?**]. The popcorn mechanism implemented in models with independent string fragmentation, like the DPM, is insufficient to explain this enhanced baryon stopping. New DPM diagrams were proposed by Kharzeev [**?**], Capella and Kopeliovich [**?**]. DPMJET-II.5 implements these diagrams and obtains an improved agreement with the net-baryon distributions in nuclear collisions.

For some light nuclei, the Glauber model calculations have been modified and the Woods–Saxon nuclear densities are replaced by parameterizations that agree better with the data. Measured nuclear radii are used [**?**] instead of the nuclear radii given by the previous parameterization. The new option XSECNUC is added to calculate a table of total, elastic, quasi-elastic, and perturbative QCD cross-sections using a modified version of the routine XSGLAU adopted from DTUNUC-II [**?**, **?**]. These changes lead to improved agreement with measured nuclear cross-sections, such as the p–Air cross-sections from cosmic-ray data.

DPMJET-II.3/4 was extended to higher energies by calculating mini-jet production using as a default the GRVLO94 parton distributions [**?**]. They are replaced in DPMJET-II.5 by the more recent GRVLO98 [**?**], which describe the most recent HERA data.

The present version of the model DPMJET-II.5 allows the inclusion or exclusion of single-diffractive events in nucleus–nucleus collisions or to sample only single-diffractive events. The diffractive cross-section in hadron–nucleus and nucleus–nucleus collisions is calculated by Monte Carlo.

3.4.4 Generators for specific studies

MEVSIM

MEVSIM [**?**] was developed for the STAR experiment to quickly produce a large number of A–A collisions for some specific needs – initially for HBT studies and for testing of reconstruction and analysis software. However, since the user is able to generate specific signals, it was extended to flow and event-by-event fluctuation analysis. A detailed description of MEVSIM can be found in Ref. [**?**].

MEVSIM generates particle spectra according to a momentum model chosen by the user. The main input parameters are: types and numbers of generated particles, momentum-distribution model, reaction-plane and azimuthal-anisotropy coefficients, multiplicity fluctuation, number of generated events, etc. The momentum models include factorized p_T and rapidity distributions, non-expanding and expanding thermal sources, arbitrary distributions in y and p_T and others. The reaction plane and azimuthal anisotropy is defined by the Fourier coefficients (maximum of six) including directed and elliptical flow. Resonance production can also be introduced.

User can define the acceptance intervals, avoiding generating particles that do not enter the detector. The generation time is negligible compared to the analysis time.

MEVSIM was originally written in FORTRAN. It was later integrated into AliRoot. A complete description of the AliRoot implementation of MEVSIM can be found on the web page (http://home.cern.ch/∼radomski).

GeVSim

GeVSim [**?**] is a fast and easy-to-use Monte Carlo event generator implemented in AliRoot. It can provide events of similar type configurable by the user according to the specific needs of a simulation project, in particular, that of flow and event-by-event fluctuation studies. It was developed to facilitate detector performance studies and for the test of algorithms. GeVSim can also be used to generate signal-free events to be processed by afterburners, for example HBT processor.

GeVSim is based on the MevSim [**?**] event generator developed for the STAR experiment. The latter was written in FORTRAN, while the former is written in C++ integrated into the ROOT environment. MevSim has been interfaced to AliRoot inside the `TMevSim` package [**?**] in order to maintain it as a common code with the STAR offline project. Since interfacing FORTRAN code makes the

design complicated and the maintenance more difficult a new package, GeVSim, written purely in C++, was developed.

GeVSim generates a list of particles by randomly sampling a distribution function. The parameters of single-particle spectra and their event-by-event fluctuations are explicitly defined by the user. Single-particle transverse-momentum and rapidity spectra can be either selected from a menu of four predefined distributions, the same as in MevSim, or provided by user.

Flow can be easily introduced into simulated events. The parameters of the flow are defined separately for each particle type and can be either set to a constant value or parameterized as a function of transverse momentum and rapidity. Two parameterizations of elliptic flow based on results obtained by RHIC experiments are provided.

GeVSim also has extended possibilities for simulating of event-by-event fluctuations. The model allows fluctuations following an arbitrary analytically defined distribution in addition to the Gaussian distribution provided by MevSim. It is also possible to systematically alter a given parameter to scan the parameter space in one run. This feature is useful when analyzing performance with respect to, for example, multiplicity or event-plane angle.

The current status and further development of GeVSim code and documentation can be found in Ref. [**?**].

HBT processor

Correlation functions constructed with the data produced by MEVSIM or any other event generator are normally flat in the region of small relative momenta. The HBT-processor afterburner introduces two particle correlations to the set of generated particles. It shifts the momentum of each particle so that the correlation function of a selected model is reproduced. The imposed correlation effects due to Quantum Statistics (QS) and Coulomb Final State Interactions (FSI) do not affect the single-particle distributions and multiplicities. The event structures before and after passing through the HBT processor are identical. Thus, the event reconstruction procedure with and without correlations is also identical. However, the track reconstruction efficiency, momentum resolution and particle identification need not to be, since correlated particles have a special topology at small relative velocities. We can thus verify the influence of various experimental factors on the correlation functions.

The data structure of reconstructed events with correlations is the same as the structure of real events. Therefore, the specific correlation-analysis software developed at the simulation stage can be directly applied to real data.

The method, proposed by L. Ray and G.W. Hoffmann [**?**] is based on random shifts of the particle three-momentum within a confined range. After each shift, a comparison is made with correlation functions resulting from the assumed model of the space–time distribution and with the single-particle spectra which

should remain unchanged. The shift is kept if the χ^2-test shows better agreement. The process is iterated until satisfactory agreement is achieved. In order to construct the correlation function, a reference sample is made by mixing particles from some consecutive events. Such a method has an important impact on the simulations when at least two events must be processed simultaneously.

Some specific features of this approach are important for practical use:

- the HBT processor can simultaneously generate correlations of up to two particle types (e.g. positive and negative pions). Correlations of other particles can be added subsequently.
- the form of the correlation function has to be parameterized analytically. One and three dimensional parameterizations are possible.
- a static source is usually assumed. Dynamical effects, related to expansion or flow, can be simulated in a stepwise form by repeating simulations for different values of the space–time parameters associated with different kinematic intervals.
- Coulomb effects may be introduced by one of the three approaches: Gamow factor, experimentally modified Gamow correction and integrated Coulomb wave functions for discrete values of the source radii.
- Strong interactions are not implemented.

The HBT processor was written in FORTRAN but is now implemented in AliRoot. The detailed description can be found elsewhere [**?**].

Generator for e^+e^- pairs in Pb–Pb collisions

In addition to strong interactions of heavy ions in central and peripheral collisions, ultra-peripheral collisions of ions give rise to coherent, mainly electromagnetic, interactions among which the dominant process is is the (multiple) e^+e^--pair production [**?**]

$$AA \rightarrow AA + n(e^+e^-), \tag{1}$$

where n is the pair multiplicity. Most electron–positron pairs are produced into the very forward direction escaping the experiment. However, for Pb–Pb collisions at the LHC the cross-section of this process, about 230 kb, is enormous. A sizable fraction of pairs produced with large-momentum transfer can contribute to the hit rate in the forward detectors increasing the occupancy or trigger rate. In order to study this effect an event generator for e^+e^--pair production has been implemented in the AliRoot framework [**?**]. The class `TEpEmGen` is a realisation of the `TGenerator` interface for external generators and wraps the FORTRAN code used to calculate the differential cross-section. `AliGenEpEmv1` derives from `AliGenerator` and uses the external generator to put the pairs on the AliRoot particle stack.

3.4.5 Combination of generators: AliGenCocktail

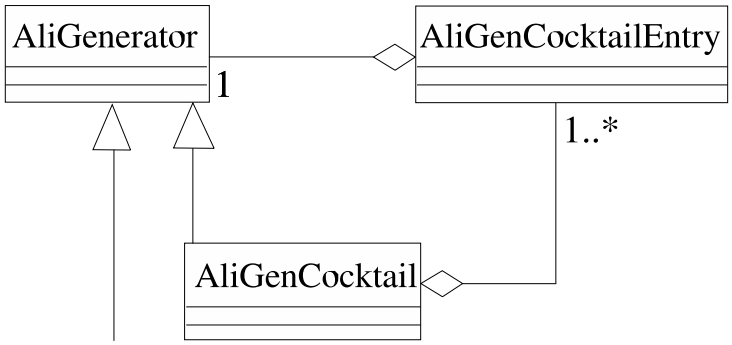

Figure 6: The `AliGenCocktail` generator is a realization of `AliGenerator` which does not generate particles itself but delegates this task to a list of objects of type `AliGenerator` that can be connected as entries (`AliGenCocktailEntry`) at run time. In this way different physics channels can be combined in one event.

Here is an example of cocktail, used for studies in the TRD detector:

```
// The cocktail generator
AliGenCocktail *gener  = new AliGenCocktail();

// Phi meson (10 particles)
AliGenParam *phi =
  new AliGenParam(10,new AliGenMUONlib(),AliGenMUONlib::kPhi,"Vogt PbPb");
phi->SetPtRange(0, 100);
phi->SetYRange(-1., +1.);
phi->SetForceDecay(kDiElectron);

// Omega meson (10 particles)
AliGenParam *omega =
  new AliGenParam(10,new AliGenMUONlib(),AliGenMUONlib::kOmega,"Vogt PbPb");
omega->SetPtRange(0, 100);
omega->SetYRange(-1., +1.);
omega->SetForceDecay(kDiElectron);

// J/psi
AliGenParam *jpsi = new AliGenParam(10,new AliGenMUONlib(),
```

```
                    AliGenMUONlib::kJpsiFamily,"Vogt PbPb");
jpsi->SetPtRange(0, 100);
jpsi->SetYRange(-1., +1.);
jpsi->SetForceDecay(kDiElectron);

// Upsilon family
AliGenParam *ups = new AliGenParam(10,new AliGenMUONlib(),
   AliGenMUONlib::kUpsilonFamily,"Vogt PbPb");
ups->SetPtRange(0, 100);
ups->SetYRange(-1., +1.);
ups->SetForceDecay(kDiElectron);

// Open charm particles
AliGenParam *charm = new AliGenParam(10,new AliGenMUONlib(),
             AliGenMUONlib::kCharm,"central");
charm->SetPtRange(0, 100);
charm->SetYRange(-1.5, +1.5);
charm->SetForceDecay(kSemiElectronic);

// Beauty particles: semi-electronic decays
AliGenParam *beauty = new AliGenParam(10,new AliGenMUONlib(),
      AliGenMUONlib::kBeauty,"central");
beauty->SetPtRange(0, 100);
beauty->SetYRange(-1.5, +1.5);
beauty->SetForceDecay(kSemiElectronic);

// Beauty particles to J/psi ee
AliGenParam *beautyJ = new AliGenParam(10, new AliGenMUONlib(),
       AliGenMUONlib::kBeauty,"central");
beautyJ->SetPtRange(0, 100);
beautyJ->SetYRange(-1.5, +1.5);
beautyJ->SetForceDecay(kBJpsiDiElectron);

// Adding all the components of the cocktail
gener->AddGenerator(phi,"Phi",1);
gener->AddGenerator(omega,"Omega",1);
gener->AddGenerator(jpsi,"J/psi",1);
gener->AddGenerator(ups,"Upsilon",1);
gener->AddGenerator(charm,"Charm",1);
gener->AddGenerator(beauty,"Beauty",1);
gener->AddGenerator(beautyJ,"J/Psi from Beauty",1);
```

```
// Settings, common for all components
gener->SetOrigin(0, 0, 0);    // vertex position
gener->SetSigma(0, 0, 5.3);   // Sigma in (X,Y,Z) (cm) on IP position
gener->SetCutVertexZ(1.);     // Truncate at 1 sigma
gener->SetVertexSmear(kPerEvent);
gener->SetTrackingFlag(1);
gener->Init();
```

3.4.6 Afterburner processors

Flow afterburner

Azimuthal anisotropies, especially elliptic flow, carry unique information about collective phenomena and consequently are important for the study of heavy-ion collisions. Additional information can be obtained studying different heavy-ion observables, especially jets, relative to the event plane. Therefore it is necessary to evaluate the capability of ALICE to reconstruct the event plane and study elliptic flow.

Since there is not a well understood microscopic description of the flow effect it cannot be correctly simulated by microscopic event generators. Therefore, to generate events with flow the user has to use event generators based on macroscopic models, like GeVSim [**?**] or an afterburner which can generate flow on top of events generated by event generators based on the microscopic description of the interaction. In the AliRoot framework such a flow afterburner is implemented.

The algorithm to apply azimuthal correlation consists in shifting the azimuthal coordinates of the particles. The transformation is given by [**?**]:

$$\varphi \rightarrow \varphi' = \varphi + \Delta\varphi$$

$$\Delta\varphi = \sum_n \frac{-2}{n} v_n \left(p_t, y\right) \sin n \times \left(\varphi - \psi\right)$$

where $v_n(p_t, y)$ is the flow coefficient to be obtained, n is the harmonic number and ψ is the event-plane angle. Note that the algorithm is deterministic and does not contain any random numbers generation.

The value of the flow coefficient can be either constant or parameterized as a function of transverse momentum and rapidity. Two parameterizations of elliptic flow are provided as in GeVSim.

```
AliGenGeVSim* gener = new AliGenGeVSim(0);

mult = 2000; // Mult is the number of charged particles in |eta| < 0.5
```

```
vn = 0.01;   // Vn

Float_t sigma_eta  = 2.75; // Sigma of the Gaussian dN/dEta
Float_t etamax     = 7.00; // Maximum eta

// Scale from multiplicity in |eta| < 0.5 to |eta| < |etamax|
Float_t mm = mult * (TMath::Erf(etamax/sigma_eta/sqrt(2.)) /
                     TMath::Erf(0.5/sigma_eta/sqrt(2.)));

// Scale from charged to total multiplicity
mm *= 1.587;

// Define particles

// 78% Pions (26% pi+, 26% pi-, 26% p0)              T = 250 MeV
AliGeVSimParticle *pp =
  new AliGeVSimParticle(kPiPlus,  1, 0.26 * mm, 0.25, sigma_eta) ;
AliGeVSimParticle *pm =
  new AliGeVSimParticle(kPiMinus, 1, 0.26 * mm, 0.25, sigma_eta) ;
AliGeVSimParticle *p0 =
  new AliGeVSimParticle(kPi0,     1, 0.26 * mm, 0.25, sigma_eta) ;

// 12% Kaons (3% K0short, 3% K0long, 3% K+, 3% K-)   T = 300 MeV
AliGeVSimParticle *ks =
  new AliGeVSimParticle(kK0Short, 1, 0.03 * mm, 0.30, sigma_eta) ;
AliGeVSimParticle *kl =
  new AliGeVSimParticle(kK0Long,  1, 0.03 * mm, 0.30, sigma_eta) ;
AliGeVSimParticle *kp =
  new AliGeVSimParticle(kKPlus,   1, 0.03 * mm, 0.30, sigma_eta) ;
AliGeVSimParticle *km =
  new AliGeVSimParticle(kKMinus,  1, 0.03 * mm, 0.30, sigma_eta) ;

// 10% Protons / Neutrons (5% Protons, 5% Neutrons)  T = 250 MeV
AliGeVSimParticle *pr =
  new AliGeVSimParticle(kProton,  1, 0.05 * mm, 0.25, sigma_eta) ;
AliGeVSimParticle *ne =
  new AliGeVSimParticle(kNeutron, 1, 0.05 * mm, 0.25, sigma_eta) ;

// Set Elliptic Flow properties

Float_t pTsaturation = 2. ;
```

```
pp->SetEllipticParam(vn,pTsaturation,0.) ;
pm->SetEllipticParam(vn,pTsaturation,0.) ;
p0->SetEllipticParam(vn,pTsaturation,0.) ;
pr->SetEllipticParam(vn,pTsaturation,0.) ;
ne->SetEllipticParam(vn,pTsaturation,0.) ;
ks->SetEllipticParam(vn,pTsaturation,0.) ;
kl->SetEllipticParam(vn,pTsaturation,0.) ;
kp->SetEllipticParam(vn,pTsaturation,0.) ;
km->SetEllipticParam(vn,pTsaturation,0.) ;

// Set Direct Flow properties

pp->SetDirectedParam(vn,1.0,0.) ;
pm->SetDirectedParam(vn,1.0,0.) ;
p0->SetDirectedParam(vn,1.0,0.) ;
pr->SetDirectedParam(vn,1.0,0.) ;
ne->SetDirectedParam(vn,1.0,0.) ;
ks->SetDirectedParam(vn,1.0,0.) ;
kl->SetDirectedParam(vn,1.0,0.) ;
kp->SetDirectedParam(vn,1.0,0.) ;
km->SetDirectedParam(vn,1.0,0.) ;

// Add particles to the list

gener->AddParticleType(pp) ;
gener->AddParticleType(pm) ;
gener->AddParticleType(p0) ;
gener->AddParticleType(pr) ;
gener->AddParticleType(ne) ;
gener->AddParticleType(ks) ;
gener->AddParticleType(kl) ;
gener->AddParticleType(kp) ;
gener->AddParticleType(km) ;

// Random Ev.Plane

TF1 *rpa = new TF1("gevsimPsiRndm","1", 0, 360);

gener->SetPtRange(0., 9.) ; // Used for bin size in numerical integration
gener->SetPhiRange(0, 360);
```

```
gener->SetOrigin(0, 0, 0);    // vertex position
gener->SetSigma(0, 0, 5.3);   // Sigma in (X,Y,Z) (cm) on IP position
gener->SetCutVertexZ(1.);     // Truncate at 1 sigma
gener->SetVertexSmear(kPerEvent);
gener->SetTrackingFlag(1);
gener->Init()
```

3.5 Particle transport

3.5.1 Description of structure elements and detectors

The front and small-angle absorber region are very detailed due to their importance to the muon spectrometer. The simulation has been instrumental in optimizing their design to save costs without a negative impact on the physics.

The L3 and dipole magnets are described both in their material distributions and their magnetic fields. The latter also includes the interference between the two fields. The field distributions are described by three independent maps for 0.2, 0.4 and $0.5\,\mathrm{T}$ solenoid L3 magnetic field strengths. Alternatively, it is possible to use simple parameterizations of the fields, i.e. constant solenoidal field in the barrel and a dipole field varying along z for the muon spectrometer.

The space frame, supporting the barrel detectors, is described according to its final design which may incorporate the recently proposed Electromagnetic Calorimeter [**?**] if it is accepted.

The design of the ALICE beam pipe has been finalized. All elements that could limit the detector performance (pumps, bellows, flanges) are represented in the simulation.

Most of the detectors have both a detailed version of the geometry, used to study their performance, and a coarse version that provides the correct material budget, with minimal details, to study the influence on other detectors without any detector response.

For some detectors, different versions corresponding to different geometry options are selectable via the input C++ script at run time. In the following we give some examples. We remind the reader that some of this information is still subject to rapid evolution.

Both a detailed and a coarse geometry are available for the ITS. The detailed geometry of the ITS is very complicated and crucially affects the evaluation of impact parameter and electron bremsstrahlung. On the other hand, simulation of the coarse geometry is much faster when ITS hits are not needed.

Attention has been devoted to the correct simulation of detector response and already very sophisticated effects can already be studied and optimized via Ali-

Root simulation.

Three configurations are available for the TPC. Version 0 is the coarse geometry, without any sensitive element specified. It is used for the material budget studies and is the version of interest for the outer detectors. Version 1 is the geometry version for the Fast Simulator. The sensitive volumes are thin gaseous strips placed in the Small (S) and Large (L) sectors at the pad-row centers. The hits are produced whenever a track crosses the sensitive volume (pad-row). The energy loss is not taken into account. Version 2 is the geometry version for the slow simulator. The sensitive volumes are S and L sectors. The user can specify either all sectors or only a few of them, up to 6 S- and 12 L-sectors. The hits are produced in every ionizing collision. The transport step is calculated for every collision from an exponential distribution. The energy loss is calculated from an $1/E^2$ distribution and the response is parameterized by a Mathieson distribution.

The TRD geometry is now quite complete, including the correct material budget for electronics and cooling pipes. The full response and digitization are implemented allowing studies of open questions such as the number of time-bins, the 9- or 10-bit ADC, the gas and electronics gain, the drift velocity, and maximum Lorentz angle. The transition-radiation photon yield is approximated by an analytical solution for a foil stack, with adjustment of the yield for a real radiator, including foam and fiber layers from test beam data. This is quite a challenging detector to simulate, as both normal energy loss in the gas and absorption of transition-radiation photons have to be taken into account.

During the signal generation several effects are taken into account: diffusion, 1-dimensional pad response, gas gain and gain fluctuations, electronics gain and noise, as well as conversion to ADC values. Absorption and $\mathbf{E} \times \mathbf{B}$ effects will be introduced.

A detailed study of the background coming from slow neutron capture in Xe gas has been performed [**?**]. The spectra of photons emitted after neutron capture are not present in standard neutron-reaction databases. An extensive literature search has been necessary in order to simulate them. The resulting code is now part of the FLUKA Monte Carlo.

The TOF detector covers a cylindrical surface of polar acceptance $|\theta - 90^\circ| < 45^\circ$. Its total weight is 25 tons and it covers an area of about 160 m^2 with about $160\,000$ total readout channels (pads) and an intrinsic resolution of $60\mathrm{ns}$. It has a modular structure corresponding to 18 sectors in φ and to 5 segments in z. All modules have the same width of 128 cm and increasing lengths, adding up to an overall TOF barrel length of 750 cm.

Inside each module the strips are tilted, thus minimizing the number of multiple partial-cell hits due to the obliqueness of the incidence angle. The double stack-strip arrangement, the cooling tubes, and the materials for electronics have been described in detail. During the development of the TOF design several dif-

ferent geometry options have been studied, all highly detailed.

The HMPID detector also poses a challenge in the simulation of the Čerenkov effect and the secondary emission of feedback photons. A detailed simulation has been introduced for all these effects and has been validated both by test-beam data and with the ALICE RICH prototype that has been operating in the STAR experiment.

The PHOS has also been simulated in detail. The geometry includes the Charged Particle Veto (CPV), crystals (EMC), readout (PIN or APD) and support structures. Hits record the energy deposition in one CPV–EMC cell per entering particle. In the digits the contribution from all particles per event are summed up and noise is added.

The simulation of the ZDC in AliRoot requires transport of spectator nucleons with Fermi spread, beam divergence and crossing angle for over 100 m. The HIJING generator is used for these studies taking into account the correlations with transverse energy and multiplicity.

The muon spectrometer is composed of 5 tracking stations and two trigger stations. For stations 1–2 a conservative material distribution is adopted, while for stations 3–5 and for the trigger stations a detailed geometry is implemented. Supporting frames and support structures are still coarse or missing but they are not very important in the simulation of the signal. The muon chambers have a complicated segmentation that has been implemented during the signal generation via a set of virtual classes. This allows changing the segmentation without modifying the geometry.

Summable digits (pad hits) are generated taking into account the Mathieson formalism for charge distribution, while work is ongoing on the angular dependence, Lorentz angle and charge correlation.

The complex T0–FMD–V0–PMD forward detector system is still under development and optimization. There are several options provided to study their performance.

ALICE geometry and generation of simulated data is in place to allow full event reconstruction including the main tracking devices. The framework allows comparison with test-beam data that has already been often performed. The early availability of a complete simulation has been an important point for the development of reconstruction and analysis code and user interfaces, now the focus of the development.

3.5.2 TGeo essential information

A detailed description of the Root geometry package is available in the Root User's Guide[**?**]. Several examples can be found in $ROOTSYS/tutorials, among them assembly.C, csgdemo.C, geodemo.C, nucleus.C, rootgeom.C, etc. Here we

show a simple usage for export/import of the ALICE geometry and for check for overlaps and extrusions:

```
aliroot
root [0] gAlice->Init()
root [1] gGeoManager->Export("geometry.root")
root [2] .q
aliroot
root [0] TGeoManager::Import("geometry.root")
root [1] gGeoManager->CheckOverlaps()
root [2] gGeoManager->PrintOverlaps()
root [3] new TBrowser
# Now you can navigate in Geometry->Illegal overlaps
# and draw each overlap (double click on it)
```

3.5.3 Visualization

Below we show an example of VZERO visualization using the Root geometry package:

```
aliroot
root [0] gAlice->Init()
root [1] TGeoVolume *top = gGeoManager->GetMasterVolume()
root [2] Int_t nd = top->GetNdaughters()
root [3] for (Int_t i=0; i<nd; i++) \
         top->GetNode(i)->GetVolume()->InvisibleAll()
root [4] TGeoVolume *v0ri = gGeoManager->GetVolume("V0RI")
root [5] TGeoVolume *v0le = gGeoManager->GetVolume("V0LE")
root [6] v0ri->SetVisibility(kTRUE);
root [7] v0ri->VisibleDaughters(kTRUE);
root [8] v0le->SetVisibility(kTRUE);
root [9] v0le->VisibleDaughters(kTRUE);
root [10] top->Draw();
```

3.5.4 Simulation of detector response

Much of the activity described in this report is a large virtual experiment where we generate thousands of events and Analise them in order to produce the results presented. This has the objective of studying in detail the ALICE physics capabilities, to clarify the physics goals of the experiment, and of verifying the functionality of our software framework from (simulated) raw data to physics.

To carry out this double objective, it is important to have a high-quality and reliable simulation. One of the most common programs for full detector simulation is GEANT 3 which, however, is a 20-year old FORTRAN program officially frozen since 1993. We are waiting for GEANT 4 to become available for production for the LHC and we also intend to evaluate FLUKA as a full detector simulation program. Therefore we decided to build an environment that could profit from the maturity and solidity of GEANT 3 and, at the same time, protect the investment in the user code when moving to a new Monte Carlo. Combining immediate needs and long term requirements into a single framework we have wrapped the GEANT 3 code in a C++ class (`TGeant3`) and we have introduced a Virtual Monte Carlo abstract interface in AliRoot, described in the next section. This has proved very satisfactory as we have been able to assure coherence of the whole simulation process:

- event generation of final-state particles. The collision is simulated by a physics generator code or a parameterization and the final-state particles are fed to the transport program.
- particle transport. The particles emerging from the interaction of the beam particles are transported in the material of the detector, simulating their interaction with it, and the energy deposition that generates the detector response (hits).
- signal generation and detector response. During this phase the detector response is generated from the energy deposition of the particles traversing it. This is the ideal detector response, before the conversion to digital signal and the formatting of the front-end electronics is applied.
- Digitization. The detector response is digitized and formatted according the the output of the front-end electronics and the data acquisition system. The results should resemble closely the real data that will be produced by the detector.
- fast simulation. The detector response is simulated via appropriate parameterizations or other techniques that do not require the full particle transport.

3.5.5 Control of physics processes

The control of the physics processes can be done from the configuration file. The relevant part is shown below:

```
// STEERING parameters FOR ALICE SIMULATION
// Specify event type to be transported through the ALICE setup
```

```
// All positions are in cm, angles in degrees, and P and E in GeV
// For the details see the GEANT 3 manual

// Swithch on/off the physics processes
gMC->SetProcess("DCAY",1);
gMC->SetProcess("PAIR",1);
gMC->SetProcess("COMP",1);
gMC->SetProcess("PHOT",1);
gMC->SetProcess("PFIS",0);
gMC->SetProcess("DRAY",0);
gMC->SetProcess("ANNI",1);
gMC->SetProcess("BREM",1);
gMC->SetProcess("MUNU",1);
gMC->SetProcess("CKOV",1);
gMC->SetProcess("HADR",1);
gMC->SetProcess("LOSS",2);
gMC->SetProcess("MULS",1);
gMC->SetProcess("RAYL",1);

// Set the transport package cuts
Float_t cut = 1.e-3;        // 1MeV cut by default
Float_t tofmax = 1.e10;

gMC->SetCut("CUTGAM", cut);
gMC->SetCut("CUTELE", cut);
gMC->SetCut("CUTNEU", cut);
gMC->SetCut("CUTHAD", cut);
gMC->SetCut("CUTMUO", cut);
gMC->SetCut("BCUTE",  cut);
gMC->SetCut("BCUTM",  cut);
gMC->SetCut("DCUTE",  cut);
gMC->SetCut("DCUTM",  cut);
gMC->SetCut("PPCUTM", cut);
gMC->SetCut("TOFMAX", tofmax);
```

In addition one may change the default parameters for each detector in the file $ALICE_ROOT/data/galice.cuts.

3.5.6 Particle decays

We use Pythia to carry one particle decays during the transport. The default decay channels can be seen in the following way:

```
aliroot
root [0] AliPythia * py = AliPythia::Instance()
root [1] py->Pylist(12); >> decay.list
```

The file decay.list will contain the list of particles decays available in Pythia. Now if we want to force the decay $\Lambda^0 \rightarrow p\pi^-$, the following lines should be included in the Config.C before we register the decayer:

```
  AliPythia * py = AliPythia::Instance();
  py->SetMDME(1059,1,0);
  py->SetMDME(1060,1,0);
  py->SetMDME(1061,1,0);
```

where 1059,1060 and 1061 are the indexes of the decay channel (from decay.list above) we want to switch off.

3.5.7 Examples

Fast simulation

This example is taken from the macro $ALICE_ROOT/FASTSIM/fastGen.C. It shows how one can create a Kinematics tree which later can be used as input for the particle transport.

```
void fastGen(Int_t nev = 1, char* filename = "galice.root") {
//  Runloader

    AliRunLoader* rl = AliRunLoader::Open("galice.root","FASTRUN","recreate");

    rl->SetCompressionLevel(2);
    rl->SetNumberOfEventsPerFile(nev);
    rl->LoadKinematics("RECREATE");
    rl->MakeTree("E");
    gAlice->SetRunLoader(rl);

//  Create stack
    rl->MakeStack();
    AliStack* stack      = rl->Stack();
```

```
//  Header
    AliHeader* header = rl->GetHeader();

//  Create and initialize Generator

    AliGenerator *gener = new AliGenPythia(1);
    gener->SetVertexSmear(kPerEvent); //   vertex position and smearing
    gener->SetStrucFunc(kGRVHO);      //   structure function
    gener->SetProcess(kPyJets);       //   selected process(jets)
    gener->SetEnergyCMS(5500.);       //   Center of mass energy
    gener->SetPtHard(50.,50.2);       //   Pt transfer of the hard scattering
    gener->Init();
    gener->SetStack(stack);

//                        Event Loop

    Int_t iev;

    for (iev = 0; iev < nev; iev++) {

printf("\n \n Event number %d \n \n", iev);

//  Initialize event
header->Reset(0,iev);
rl->SetEventNumber(iev);
stack->Reset();
rl->MakeTree("K");

//  Generate event
gener->Generate();
//  ``Analysis''
Int_t npart = stack->GetNprimary();
printf("Analyse %d Particles\n", npart);
for (Int_t part=0; part<npart; part++) {
    TParticle *MPart = stack->Particle(part);
    Int_t mpart  = MPart->GetPdgCode();
    printf("Particle %d\n", mpart);
}

//  Finish event
```

```
header->SetNprimary(stack->GetNprimary());
header->SetNtrack(stack->GetNtrack());

//      I/O
stack->FinishEvent();
header->SetStack(stack);
rl->TreeE()->Fill();
rl->WriteKinematics("OVERWRITE");

    } // event loop
//
//                          Termination
//  Generator
    gener->FinishRun();
//  Stack
    stack->FinishRun();
//  Write file
    rl->WriteHeader("OVERWRITE");
    gener->Write();
    rl->Write();
}
```

Reading of kinematics tree as input for the particle transport

We suppose that the macro fastGen.C above has been used to generate the corresponding sent of files: galice.root and Kinematics.root, and that they are stored in a separate subdirectory, for example kine. Then the following code in Config.C fill read the set of files and put them in the stack for transport:

```
    AliGenExtFile *gener = new AliGenExtFile(-1);

    gener->SetMomentumRange(0,14000);
    gener->SetPhiRange(0.,360.);
    gener->SetThetaRange(45,135);
    gener->SetYRange(-10,10);
    gener->SetOrigin(0, 0, 0);  //vertex position
    gener->SetSigma(0, 0, 5.3);   //Sigma in (X,Y,Z) (cm) on IP position

    AliGenReaderTreeK * reader = new AliGenReaderTreeK();
    reader->SetFileName("../galice.root");

    gener->SetReader(reader);
```

```
gener->SetTrackingFlag(1);

gener->Init();
```

Usage of different generators

A lot of examples are available in $ALICE_ROOT/macros/Config_gener.C. The correspondent part can be extracted and placed in the relevant Config.C file.

4 Reconstruction

4.1 Reconstruction Framework

This chapter focuses on the reconstruction framework from the (detector) software developers point of view.

Wherever it is not specified explicitly as different, we refer to the 'global ALICE coordinate system'. It is a right-handed coordinate system with the z axis coinciding with the beam-pipe axis and going in the direction opposite to the muon arm, the y axis going up, and the origin of coordinates defined by the intersection point of the z axis and the central-membrane plane of TPC.

We also use the following terms:

- *Digit*: This is a digitized signal (ADC count) obtained by a sensitive pad of a detector at a certain time.
- *Cluster*: This is a set of adjacent (in space and/or in time) digits that were presumably generated by the same particle crossing the sensitive element of a detector.
- Reconstructed *space point*: This is the estimation of the position where a particle crossed the sensitive element of a detector (often, this is done by calculating the center of gravity of the 'cluster').
- Reconstructed *track*: This is a set of five parameters (such as the curvature and the angles with respect to the coordinate axes) of the particle's trajectory together with the corresponding covariance matrix estimated at a given point in space.

The input to the reconstruction framework are digits in root tree format or raw data format. First a local reconstruction of clusters is performed in each detector. Then vertexes and tracks are reconstructed and particles types are identified. The output of the reconstruction is the Event Summary Data (ESD). The

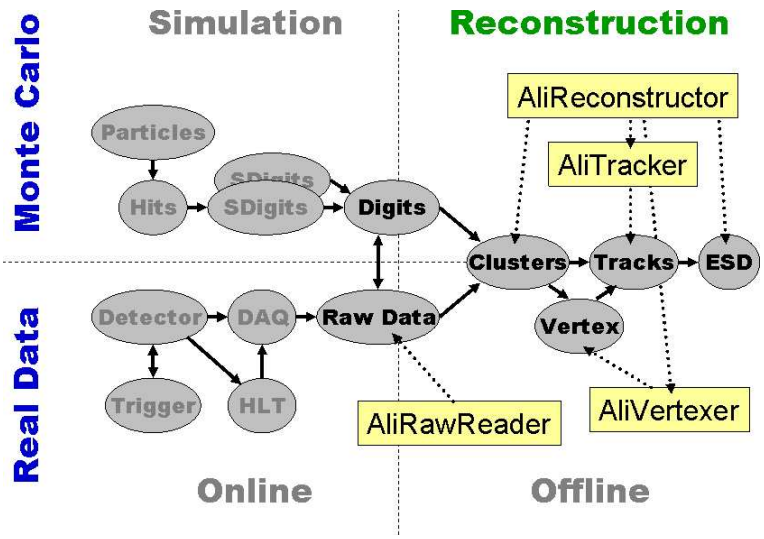

Figure 7: Reconstruction framework.

`AliReconstruction` class provides a simple user interface to the reconstruction framework which is explained in the source code and.

Requirements and Guidelines

The following list contains items that are considered important. The importance varies from essential to only nice-to-have and may depend on the personal point of view. The list and the importance of the items will change in time with the improved understanding of our needs:

- the prime goal is to provide the data that is needed for a physics analysis.
- it should be aimed for high efficiency, purity and resolution.
- the user should have an easy to use interface to extract the required information.
- the reconstruction code should be efficient but also maintainable.
- the reconstruction should be as flexible as possible. It should be possible to do the reconstruction in one detector even in the case that other detectors are not operational. To achieve such a flexibility each detector module should be able to
  - find tracks starting from seeds provided by another detector (external seeding),
  - find tracks without using information from other detectors (internal seeding),

  - find tracks from external seeds and add tracks from internal seeds
  - and propagate tracks through the detector using the already assigned clusters in inward and outward direction.

- where it is appropriate, common (base) classes should be used in the different reconstruction modules.
- the interdependencies between the reconstruction modules should be minimized. If possible the exchange of information between detectors should be done via a common track class.
- the chain of reconstruction program(s) should be callable and steerable in an easy way.
- there should be no assumptions on the structure or names of files or on the number or order of events.
- each class, data member and method should have a correct, precise and helpful html documentation.

AliReconstructor

The interface from the steering class `AliReconstruction` to the detector specific reconstruction code is defined by the base class `AliReconstructor`. For each detector there is a derived reconstructor class. The user can set options for each reconstructor in format of a string parameter which is accessible inside the reconstructor via the method GetOption.

The detector specific reconstructors are created via plugins. Therefore they must have a default constructor. If no plugin handler is defined by the user (in .rootrc), it is assumed that the name of the reconstructor for detector DET is AliDETReconstructor and that it is located in the library libDETrec.so (or libDET.so).

Input Data

If the input data is provided in format of root trees, either the loaders or directly the trees are used to access the digits. In case of raw data input the digits are accessed via a raw reader.

If a galice.root file exists, the run loader will be retrieved from it. Otherwise the run loader and the headers will be created from the raw data. The reconstruction can not work if there is no galice.root file and no raw data input.

Output Data

The clusters (rec. points) are considered as intermediate output and are stored in root trees handled by the loaders. The final output of the reconstruction is a tree with objects of type `AliESD` stored in the file AliESDs.root. This Event

Summary Data (ESD) contains lists of reconstructed tracks/particles and global event properties.

Local Reconstruction (Clusterization)

The first step of the reconstruction is the so called local reconstruction. It is executed for each detector separately and without exchanging information with other detectors. Usually the clusterization is done in this step.

The local reconstruction is invoked via the method Reconstruct of the reconstructor object. The deprecated `Reconstruct` methods with a run loader or a run loader and a raw reader as argument do the local reconstruction for all events. The event loop and the loading of the trees is handled by the `AliReconstruction` class in case of the Reconstruct methods with two trees or a raw reader and a tree as argument. Only the local reconstruction of the current event has to be performed in these two methods for the two different types of digit input data given by the first argument. The second argument provides the output tree for the clusters. The local reconstruction method is only called if the method HasLocal-Reconstruction of the reconstructor returns kTRUE.

Instead of running the local reconstruction directly on raw data, it is possible to first convert the raw data digits into a digits tree and then to call the `Reconstruct` method with a tree as input parameter. This conversion is done by the method ConvertDigits. The reconstructor has to announce that it can convert the raw data digits by returning kTRUE in the method `HasDigitConversion`.

Vertexing

The reconstruction of the primary-vertex position in ALICE is done using the information provided by the silicon pixel detectors, which constitute the two innermost layers of the ITS.

The algorithm starts with looking at the distribution of the z coordinates of the reconstructed space points in the first pixel layers. At a vertex z coordinate $z_{\rm true} = 0$ the distribution is symmetric and its centroid ($z_{\rm cen}$) is very close to the nominal vertex position. When the primary vertex is moved along the z axis, an increasing fraction of hits will be lost and the centroid of the distribution no longer gives the primary vertex position. However, for primary vertex locations not too far from $z_{\rm true} = 0$ (up to about 12 cm), the centroid of the distribution is still correlated to the true vertex position. The saturation effect at large $z_{\rm true}$ values of the vertex position ($z_{\rm true}$ =12–15 cm) is, however, not critical, since this procedure is only meant to find a rough vertex position, in order to introduce some cut along z.

To find the final vertex position, the correlation between the points z_1, z_2 in the two layers was considered. More details and performance studies are available in [**?**].

The primary vertex is reconstructed by a vertexer object derived from `AliVertexer`. After the local reconstruction was done for all detectors the vertexer method

`FindVertexForCurrentEvent` is called for each event. It returns a pointer to a vertex object of type `AliESDVertex`.

The vertexer object is created by the method `CreateVertexer` of the reconstructor. So far only the ITS is used to determine the primary vertex (`AliITSVertexerZ` class).

The precision of the primary vertex reconstruction in the bending plane required for the reconstruction of D and B mesons in pp events can be achieved only after the tracking is done. The method is implemented in `AliITSVertexerTracks`.

Each track, reconstructed in the TPC and in the ITS, is approximated with a straight line at the position of the closest approach to the nominal primary vertex position (the nominal vertex position is supposed to be known with a precision of 100–200 μm). Then, all possible track pairs (i, j) are considered and for each pair, the center $C(i, j) \equiv (x_{ij}, y_{ij}, z_{ij})$ of the segment of minimum approach between the two lines is found. The coordinates of the primary vertex are determined as:

$$x_{\rm v} = \frac{1}{N_{\rm pairs}} \sum_{i,j} x_{ij} \; ; \qquad y_{\rm v} = \frac{1}{N_{\rm pairs}} \sum_{i,j} y_{ij} \; ; \qquad z_{\rm v} = \frac{1}{N_{\rm pairs}} \sum_{i,j} z_{ij}$$

where $N_{\rm pairs}$ is the number of track pairs. This gives an improved estimate of the vertex position.

Finally, the position $\mathbf{r}_{\rm v} = (x_{\rm v}, y_{\rm v}, z_{\rm v})$ of the vertex is reconstructed minimizing the χ^2 function (see Ref. [**?**]):

$$\chi^2(\mathbf{r}_{\rm v}) = \sum_i (\mathbf{r}_{\rm v} - \mathbf{r}_i)^T \, \mathbf{V}_i^{-1} (\mathbf{r}_{\rm v} - \mathbf{r}_i), \tag{2}$$

where $\mathbf{r}_i$ is the global position of the track i (i.e. the position assigned at the step above) and $\mathbf{V}_i$ is the covariance matrix of the vector $\mathbf{r}_i$.

In order not to spoil the vertex resolution by including in the fit tracks that do not originate from the primary vertex (e.g. strange particle decay tracks), the tracks giving a contribution larger than some value $\chi^2_{\rm max}$ to the global χ^2 are removed one-by-one from the sample, until no such tracks are left. The parameter $\chi^2_{\rm max}$ was tuned, as a function of the event multiplicity, so as to obtain the best vertex resolution.

Combined Track Reconstruction The combined track reconstruction tries to accumulate the information from different detectors in order to optimize the track reconstruction performance. The result of this is stored in the combined track objects. The `AliESDTrack` class also provides the possibility to exchange information between detectors without introducing dependencies between the reconstruction modules. This is achieved by using just integer indexes pointing to the specific track objects, which on the other hand makes it possible to retrieve the full information if needed. The list of combined tracks can be kept in memory

and passed from one reconstruction module to another. The storage of the combined tracks should be done in the standard way (newIO).

The classes responsible for the reconstruction of tracks are derived from `AliTracker`. They are created by the method `CreateTracker` of the reconstructors. The reconstructed position of the primary vertex is made available to them via the method `SetVertex`. Before the track reconstruction in a detector starts the clusters are loaded from the clusters tree by the method `LoadClusters`. After the track reconstruction the clusters are unloaded by the method `UnloadClusters`.

The track reconstruction (in the barrel part) is done in three passes. The first pass consists of a track finding and fitting in inward direction in TPC and then in ITS. The virtual method Clusters2Tracks is the interface to this pass. The method for the next pass is PropagateBack. It does the track reconstruction in outward direction and is invoked for all detectors starting with the ITS. The last pass is the track refit in inward direction in order to get the track parameters at the vertex. The corresponding method RefitInward is called for TRD, TPC and ITS. All three track reconstruction methods have an AliESD object as argument which is used to exchange track information between detectors without introducing dependences between the code of the detector trackers.

Depending on the way the information is used, the tracking methods can be divided into two large groups: global methods and local methods. Each group has advantages and disadvantages.

With the global methods, all the track measurements are treated simultaneously and the decision to include or exclude a measurement is taken when all the information about the track is known. Typical algorithms belonging to this class are combinatorial methods, Hough transform, templates, conformal mappings. The advantages are the stability with respect to noise and mismeasurements and the possibility to operate directly on the raw data. On the other hand, these methods require a precise global track model. Such a track model can sometimes be unknown or does not even exist because of stochastic processes (energy losses, multiple scattering), non-uniformity of the magnetic field etc. In ALICE, global tracking methods are being extensively used in the High-Level Trigger (HLT) software. There, we are mostly interested in the reconstruction of the high-momentum tracks only, the required precision is not crucial, but the speed of the calculations is of great importance.

Local methods do not need the knowledge of the global track model. The track parameters are always estimated 'locally' at a given point in space. The decision to accept or to reject a measurement is made using either the local information or the information coming from the previous 'history' of this track. With these methods, all the local track peculiarities (stochastic physics processes, magnetic fields, detector geometry) can be naturally accounted for. Unfortunately, the local methods rely on sophisticated space point reconstruction algorithms (including

unfolding of overlapped clusters). They are sensitive to noise, wrong or displaced measurements and the precision of space point error parametrization. The most advanced kind of local track-finding methods is Kalman filtering which was introduced by P. Billoir in 1983 [**?**].

Kalman filtering is quite a general and powerful method for statistical estimations and predictions. The conditions for its applicability are the following. A certain 'system' is determined at any moment in time t_k by a state vector x_k. The state vector varies with time according to an evolution equation

$$x_k = f_k(x_{k-1}) + \epsilon_k.$$

It is supposed that f_k is a known deterministic function and ϵ_k is a random vector of intrinsic 'process noise' which has a zero mean value ($< \epsilon_k >= 0$) and a known covariance matrix ($\mathrm{cov}(\epsilon_k) = Q_k$). Generally, only some function h_k of the state vector can be observed, and the result of the observation m_k is corrupted by a 'measurement noise' δ_k:

$$m_k = h_k(x_k) + \delta_k.$$

The measurement noise is supposed to be unbiased ($< \delta_k >= 0$) and have a definite covariance matrix ($\mathrm{cov}(\delta_k) = V_k$). In many cases, the measurement function h_k can be represented by a certain matrix H_k:

$$m_k = H_k x_k + \delta_k.$$

If, at a certain time t_{k-1}, we are given some estimates of the state vector $\tilde{x}_{k-1}$ and of its covariance matrix $\tilde{C}_{k-1} = \mathrm{cov}(\tilde{x}_{k-1} - x_{k-1})$, we can extrapolate these estimates to the next time slot t_k by means of formulas (this is called 'prediction'):

$$\begin{aligned} \tilde{x}_k^{k-1} &= f_k(\tilde{x}_{k-1}) \\ \tilde{C}_k^{k-1} &= F_k \tilde{C}_{k-1} F_k^T + Q_k, \quad F_k = \frac{\partial f_k}{\partial x_{k-1}}. \end{aligned}$$

The value of the predicted χ^2 increment can be also calculated:

$$(\chi^2)_k^{k-1} = (r_k^{k-1})^T (R_k^{k-1})^{-1} r_k^{k-1}, \quad r_k^{k-1} = m_k - H_k \tilde{x}_k^{k-1}, \quad R_k^{k-1} = V_k + H_k \tilde{C}_k^{k-1} H_k^T. \tag{3}$$

The number of degrees of freedom is equal to the dimension of the vector m_k.

If at the moment t_k, together with the results of prediction, we also have the results of the state vector measurement, this additional information can be combined with the prediction results (this is called 'filtering'). As a consequence, the estimation of the state vector improves:

$$\begin{aligned} \tilde{x}_k &= \tilde{x}_k^{k-1} + K_k(m_k - H_k \tilde{x}_k^{k-1}) \\ \tilde{C}_k &= \tilde{C}_k^{k-1} - K_k H_k \tilde{C}_k^{k-1}, \end{aligned}$$

where K_k is the Kalman gain matrix $K_k = \tilde{C}_k^{k-1} H_k^T (V_k + H_k \tilde{C}_k^{k-1} H_k^T)^{-1}$.

Finally, the next formula gives us the value of the filtered χ^2 increment:

$$\chi_k^2 = (r_k)^T (R_k)^{-1} r_k, \quad r_k = m_k - H_k \tilde{x}_k, \quad R_k = V_k - H_k \tilde{C}_k H_k^T.$$

It can be shown that the predicted χ^2 value is equal to the filtered one:

$$(\chi^2)_k^{k-1} = \chi_k^2. \tag{4}$$

The 'prediction' and 'filtering' steps are repeated as many times as we have measurements of the state vector.

When applied to the track reconstruction problem, the Kalman-filter approach shows many attractive properties:

- It is a method for simultaneous track recognition and fitting.
- There is a possibility to reject incorrect space points 'on the fly', during the only tracking pass (see Eq. **??**). These incorrect points can appear as a consequence of the imperfection of the cluster finder. They may be due to noise or they may be points from other tracks accidentally captured in the list of points to be associated with the track under consideration. In the other tracking methods one usually needs an additional fitting pass to get rid of incorrectly assigned points.
- In the case of substantial multiple scattering, track measurements are correlated and therefore large matrices (of the size of the number of measured points) need to be inverted during a global fit. In the Kalman-filter procedure we only have to manipulate up to 5×5 matrices (although as many times as we have measured space points), which is much faster.
- One can handle multiple scattering and energy losses in a simpler way than in the case of global methods.
- It is a natural way to find the extrapolation of a track from one detector to another (for example from the TPC to the ITS or to the TRD).

In ALICE we require good track-finding efficiency and reconstruction precision for track down to p_{t}= 100 MeV/c. Some of the ALICE tracking detectors (ITS, TRD) have a significant material budget. Under such conditions one can not neglect the energy losses or the multiple scattering in the reconstruction. There are also rather big dead zones between the tracking detectors which complicates finding the continuation of the same track. For all these reasons, it is the Kalman-filtering approach that has been our choice for the offline reconstruction since 1994.

The reconstruction software for the ALICE central tracking detectors (the ITS, TPC and the TRD) shares a common convention on the coordinate system used. All the clusters and tracks are always expressed in some local coordinate system related to a given sub-detector (TPC sector, ITS module etc). This local coordinate system is defined as the following:

- It is a right handed-Cartesian coordinate system;
- its origin and the z axis coincide with those of the global ALICE coordinate system;
- the x axis is perpendicular to the sub-detector's 'sensitive plane' (TPC pad row, ITS ladder etc).

Such a choice reflects the symmetry of the ALICE set-up and therefore simplifies the reconstruction equations. It also enables the fastest possible transformations from a local coordinate system to the global one and back again, since these transformations become simple single rotations around the z-axis.

The reconstruction begins with cluster finding in all of the ALICE central detectors (ITS, TPC, TRD, TOF, HMPID and PHOS). Using the clusters reconstructed at the two pixel layers of the ITS, the position of the primary vertex is estimated and the track finding starts. As described later, cluster-finding as well as the track-finding procedures performed in the detectors have some different detector-specific features. Moreover, within a given detector, on account of high occupancy and a big number of overlapped clusters, the cluster finding and the track finding are not completely independent: the number and positions of the clusters are finally determined only at the track-finding step.

The general tracking strategy is the following. We start from our best tracker device, i.e. the TPC, and from the outer radius where the track density is minimal. First, the track candidates ('seeds') are found. Because of the small number of clusters assigned to a seed, the precision of its parameters is not enough to safely extrapolate it outwards to the other detectors. Instead, the tracking stays within the TPC and proceeds towards the smaller TPC radii. Whenever possible, new clusters are associated with a track candidate in a 'classical' Kalman-filter way and the track parameters are more and more refined. When all of the seeds are extrapolated to the inner limit of the TPC, the tracking in the ITS takes over. The ITS tracker tries to prolong the TPC tracks as close as possible to the primary vertex. On the way to the primary vertex, the tracks are assigned additional, precisely reconstructed ITS clusters, which also improves the estimation of the track parameters.

After all the track candidates from the TPC are assigned their clusters in the ITS, a special ITS stand-alone tracking procedure is applied to the rest of the ITS

clusters. This procedure tries to recover the tracks that were not found in the TPC because of the p_{t}cut-off, dead zones between the TPC sectors, or decays.

At this point the tracking is restarted from the vertex back to the outer layer of the ITS and then repeated towards the outer wall of the TPC. For the track that was labeled by the ITS tracker as potentially primary, several particle-mass-dependent, time-of-flight hypotheses are calculated. These hypotheses are then used for the particle identification (PID) with the TOF detector. Once the outer radius of the TPC is reached, the precision of the estimated track parameters is sufficient to extrapolate the tracks to the TRD, TOF, HMPID and PHOS detectors. Tracking in the TRD is done in a similar way to that in the TPC. Tracks are followed till the outer wall of the TRD and the assigned clusters improve the momentum resolution further. Next, the tracks are extrapolated to the TOF, HMPID and PHOS, where they acquire the PID information. Finally, all the tracks are refitted with the Kalman filter backwards to the primary vertex (or to the innermost possible radius, in the case of the secondary tracks).

The tracks that passed the final refit towards the primary vertex are used for the secondary vertex (V^0, cascade, kink) reconstruction. There is also an option to reconstruct the secondary vertexes 'on the fly' during the tracking itself. The potential advantage of such a possibility is that the tracks coming from a secondary vertex candidate are not extrapolated beyond the vertex, thus minimizing the risk of picking up a wrong track prolongation. This option is currently under investigation.

The reconstructed tracks (together with the PID information), kink, V^0 and cascade particle decays are then stored in the Event Summary Data (ESD).

More details about the reconstruction algorithms can be found in Chapter 5 of the ALICE Physics Performance Report[**?**].

Filling of ESD

After the tracks were reconstructed and stored in the `AliESD` object, further information is added to the ESD. For each detector the method `FillESD` of the reconstructor is called. Inside this method e.g. V0s are reconstructed or particles are identified (PID). For the PID a Bayesian approach is used which is explained in a note. The constants and some functions that are used for the PID are defined in the class `AliPID`.

Monitoring of Performance

For the monitoring of the track reconstruction performance the classes `AliTrackReference` are used. Objects of the second type of class are created during the reconstruction at the same locations as the `AliTrackReference` objects. So the reconstructed tracks can be easily compared with the simulated particles. This allows to study and monitor the performance of the track reconstruction in detail. The creation of the objects used for the comparison should not interfere with the reconstruction algorithm and can be switched on or off.

Several "comparison" macros permit to monitor the efficiency and the resolution of the tracking. Here is a typical usage (the simulation and the reconstruction have been done in advance):

```
aliroot
root [0] gSystem->SetIncludePath("-I$ROOTSYS/include \
                                  -I$ALICE_ROOT/include \
                                  -I$ALICE_ROOT/TPC \
                                  -I$ALICE_ROOT/ITS \
  -I$ALICE_ROOT/TOF")
root [1] .L $ALICE_ROOT/TPC/AliTPCComparison.C++
root [2] .L $ALICE_ROOT/ITS/AliITSComparisonV2.C++
root [3] .L $ALICE_ROOT/TOF/AliTOFComparison.C++
root [4] AliTPCComparison()
root [5] AliITSComparisonV2()
root [6] AliTOFComparison()
```

Another macro can be used to provide a preliminary estimate of the combined acceptance: $ALICE_ROOT/STEER/CheckESD.C.

Classes

There are many classes used in the reconstruction. Some of the listed classes are only propositions so far and for some classes there are alternative names proposed.

- `AliTrackReference`: This class is used to store the position and the momentum of a simulated particle at given locations of interest (e.g. when the particle enters or exits a detector or it decays). It is used for mainly for debugging and tuning of the tracking.

- `AliExternalTrackParams`: This class describes the status of a track in a given point. It knows the track parameters and its covariance matrix. This parameterization is used to exchange tracks between the detectors. A set of functions returning the position and the momentum of tracks in the global coordinate system as well as the track impact parameters are implemented. There is possibility to propagate the track to a given radius `PropagateTo` and `Propagate`.

- `AliKalmanTrack` and subclasses: These classes are used to find and fit tracks with the Kalman approach. The `AliKalmanTrack` fixes the interfaces and implements some common functionality. The derived classes know about the clusters assigned to the track. They also update the information in an `AliESDtrack`. The current status of the track during the track

reconstruction can be represented by an AliExternalTrackParameters. The history of the track during the track reconstruction can be stored in a list of AliExternalTrackParameters objects. The AliKalmanTrack defines the methods:

  - Double_t GetDCA(...) Returns the (weighed !) distance of closest approach between this track and the track passed as the argument.
  - Double_t MeanMaterialBudget(...) Calculate the mean material budget and material properties between two points.

- AliTracker and subclasses: The AliTracker is the base class for all the trackers in the different detectors. It fixes the interface needed to find and propagate tracks. The actual implementation is done in the derived classes.

- AliESDTrack: This class combines the information about a track from different detectors. It knows the current status of the track (AliExternalTrackParameters) and it has (non-persistent) pointers to the individual AliKalmanTrack objects from each detector which contributed to the track. It knows about some detector specific quantities like the number or bit pattern of assigned clusters, dEdx, χ^2, etc.. And it can calculate a conditional probability for a given mixture of particle species following the Bayesian approach. It defines a track label pointing to the corresponding simulated particle in case of Monte Carlo. The combined track objects are the basis for a physics analysis.

Example

The example below shows reconstruction with non-uniform magnetic field (the simulation is also done with non-uniform magnetic field by adding the following line in the Config.C: field-¿SetL3ConstField(1)). Only the barrel detectors are reconstructed, a specific TOF reconstruction has been requested, and the RAW data have been used:

```
void rec() {
  AliReconstruction reco;
  reco.SetOption("TOF","MI");

  reco.SetRunReconstruction("ITS TPC TRD TOF");
  reco.SetNonuniformFieldTracking();
  reco.SetInput("raw.root");

  TStopwatch timer;
  timer.Start();
```

```
  reco.Run();
  timer.Stop();
  timer.Print();
}
```

4.2 Event summary data

The classes which are needed to process and analyze the ESD are packed together in a standalone library (libESD.so) which can be used separately from the AliRoot framework. The main class is `AliESD`, which contains all the information needed during the physics analysis:

- fields to identify the event such as run number, event number, trigger word, version of the reconstruction, etc.;
- reconstructed ZDC energies and number of participant;
- primary vertex;
- T0 estimation of the primary vertex;
- array of ESD tracks;
- arrays of HLT tracks both from the conformal mapping and from the Hough transform reconstruction;
- array of MUON tracks;
- array of PMD tracks;
- arrays of reconstructed V^0 vertexes, cascade decays and kinks;
- indexes of the information from PHOS and EMCAL detectors in the array of the ESD tracks.

5 Analysis

5.1 Introduction

The analysis of experimental data is the final stage of event processing and it is usually repeated many times. Analysis is a very diverse activity, where the goals of each particular analysis pass may differ significantly.

The ALICE detector [**?**] is optimized for the reconstruction and analysis of heavy-ion collisions. In addition, ALICE has a broad physics programme devoted to p–p and p–A interactions.

The main points of the ALICE heavy-ion programme can be summarized as follows[**?**]:

- **global event characteristics:** particle multiplicity, centrality, energy density, nuclear stopping;
- **soft physics:** chemical composition (particle and resonance production, particle ratios and spectra, strangeness enhancement), reaction dynamics (transverse and elliptic flow, HBT correlations, event-by-event dynamical fluctuations);
- **hard probes:** jets, direct photons;
- **heavy flavors:** quarkonia, open charm and beauty production.

The p–p and p–A programme will provide, on the one hand, reference points for comparison with heavy ions. On the other hand, ALICE will also pursue genuine and detailed p–p studies. Some quantities, in particular the global characteristics of interactions, will be measured during the first days of running exploiting the low-momentum measurement and particle identification capabilities of ALICE.

The ALICE computing framework is described in details in the Computing Technical Design Report [**?**]. This article is based on Chapter 6 of the document.

5.1.1 The analysis activity

We distinguish two main types of analysis: scheduled analysis and chaotic analysis. They differ in their data access pattern, in the storage and registration of the results, and in the frequency of changes in the analysis code. The detailed definition is given in Section **??**.

In the ALICE Computing Model the analysis starts from the Event Summary Data (ESD). These are produced during the reconstruction step and contain all the information for the analysis. The size of the ESD is about one order of magnitude lower than the corresponding raw data. The analysis tasks produce Analysis Object Data (AOD) specific to a given set of physics objectives. Further passes for the specific analysis activity can be performed on the AODs, until the selection parameter or algorithms are changed.

A typical data analysis task usually requires processing of selected sets of events. The selection is based on the event topology and characteristics, and is

done by querying the tag database. The tags represent physics quantities which characterize each run and event, and permit fast selection. They are created after the reconstruction and contain also the unique identifier of the ESD file. A typical query, when translated into natural language, could look like "Give me all the events with impact parameter in <range> containing jet candidates with energy larger than <threshold>". This results in a list of events and file identifiers to be used in the consecutive event loop.

The next step of a typical analysis consists of a loop over all the events in the list and calculation of the physics quantities of interest. Usually, for each event, there is a set of embedded loops on the reconstructed entities such as tracks, V^0 candidates, neutral clusters, etc., the main goal of which is to select the signal candidates. Inside each loop a number of criteria (cuts) are applied to reject the background combinations and to select the signal ones. The cuts can be based on geometrical quantities such as impact parameters of the tracks with respect to the primary vertex, distance between the cluster and the closest track, distance of closest approach between the tracks, angle between the momentum vector of the particle combination and the line connecting the production and decay vertexes. They can also be based on kinematics quantities such as momentum ratios, minimal and maximal transverse momentum, angles in the rest frame of the combination. Particle identification criteria are also among the most common selection criteria.

The optimization of the selection criteria is one of the most important parts of the analysis. The goal is to maximize the signal-to-background ratio in case of search tasks, or another ratio (typically $\mathrm{Signal}/\sqrt{\mathrm{Signal}+\mathrm{Background}}$) in case of measurement of a given property. Usually, this optimization is performed using simulated events where the information from the particle generator is available.

After the optimization of the selection criteria, one has to take into account the combined acceptance of the detector. This is a complex, analysis-specific quantity which depends on the geometrical acceptance, the trigger efficiency, the decays of particles, the reconstruction efficiency, the efficiency of the particle identification and of the selection cuts. The components of the combined acceptance are usually parametrized and their product is used to unfold the experimental distributions or during the simulation of some model parameters.

The last part of the analysis usually involves quite complex mathematical treatments, and sophisticated statistical tools. Here one may include the correction for systematic effects, the estimation of statistical and systematic errors, etc.

5.2 Organization of the data analysis

The data analysis is coordinated by the Physics Board via the Physics Working Groups (PWGs). At present the following PWG have started their activity:

- detector performance;
- global event characteristics and soft physics (including proton–proton physics);
- hard probes: jets and direct photons;
- heavy flavors.

Scheduled analysis

The scheduled analysis typically uses all the available data from a given period, and stores and registers the results using Grid middleware. The tag database is updated accordingly. The AOD files, generated during the scheduled analysis, can be used by several subsequent analyses, or by a class of related physics tasks. The procedure of scheduled analysis is centralized and can be considered as data filtering. The requirements come from the PWGs and are prioritized by the Physics Board taking into account the available computing and storage resources. The analysis code is tested in advance and released before the beginning of the data processing.

Each PWG will require several sets of AOD per event, which are specific for one or a few analysis tasks. The creation of the AOD sets is managed centrally. The event list of each AOD set will be registered and the access to the AOD files will be granted to all ALICE collaborators. AOD files will be generated via Grid tools at different computing centers and will be stored on the corresponding storage elements. The processing of each file set will thus be done in a distributed way on the Grid. Some of the AOD sets may be quite small and would fit on a single storage element or even on one computer; in this case the corresponding tools for file replication, available in the ALICE Grid infrastructure, will be used.

Chaotic analysis

The chaotic analysis is focused on a single physics task and typically is based on the filtered data from the scheduled analysis. Each physicist also may access directly large parts of the ESD in order to search for rare events or processes. Usually the user develops the code using a small subsample of data, and changes the algorithms and criteria frequently. The analysis macros and software are tested many times on relatively small data volumes, both experimental and Monte Carlo. The output is often only a set of histograms. Such a tuning of the analysis code can be done on a local data set or on distributed data using Grid tools. The final version of the analysis will eventually be submitted to the Grid and will access large portions or even the totality of the ESDs. The results may be registered in the Grid file catalog and used at later stages of the analysis. This activity may or may

not be coordinated inside the PWGs, via the definition of priorities. The chaotic analysis is carried on within the computing resources of the physics groups.

5.3 Infrastructure tools for distributed analysis

5.3.1 gShell

The main infrastructure tools for distributed analysis have been described in Chapter 3. The actual middleware is hidden by an interface to the Grid, gShell[**?**], which provides a single working shell. The gShell package contains all the commands a user may need for file catalog queries, creation of sub-directories in the user space, registration and removal of files, job submission and process monitoring. The actual Grid middleware is completely transparent to the user.

The gShell overcomes the scalability problem of direct client connections to databases. All clients connect to the gLite[**?**] API services. This service is implemented as a pool of preforked server daemons, which serve single-client requests. The client-server protocol implements a client state which is represented by a current working directory, a client session ID and time-dependent symmetric cipher on both ends to guarantee client privacy and security. The server daemons execute client calls with the identity of the connected client.

5.3.2 PROOF – the Parallel ROOT Facility

The Parallel ROOT Facility, PROOF[**?**] has been specially designed and developed to allow the analysis and mining of very large data sets, minimizing response time. It makes use of the inherent parallelism in event data and implements an architecture that optimizes I/O and CPU utilization in heterogeneous clusters with distributed storage. The system provides transparent and interactive access to terabyte-scale data sets. Being part of the ROOT framework, PROOF inherits the benefits of a performing object storage system and a wealth of statistical and visualization tools. The most important design features of PROOF are:

- transparency – no difference between a local ROOT and a remote parallel PROOF session;
- scalability – no implicit limitations on number of computers used in parallel;
- adaptability – the system is able to adapt to variations in the remote environment.

PROOF is based on a multi-tier architecture: the ROOT client session, the PROOF master server, optionally a number of PROOF sub-master servers, and

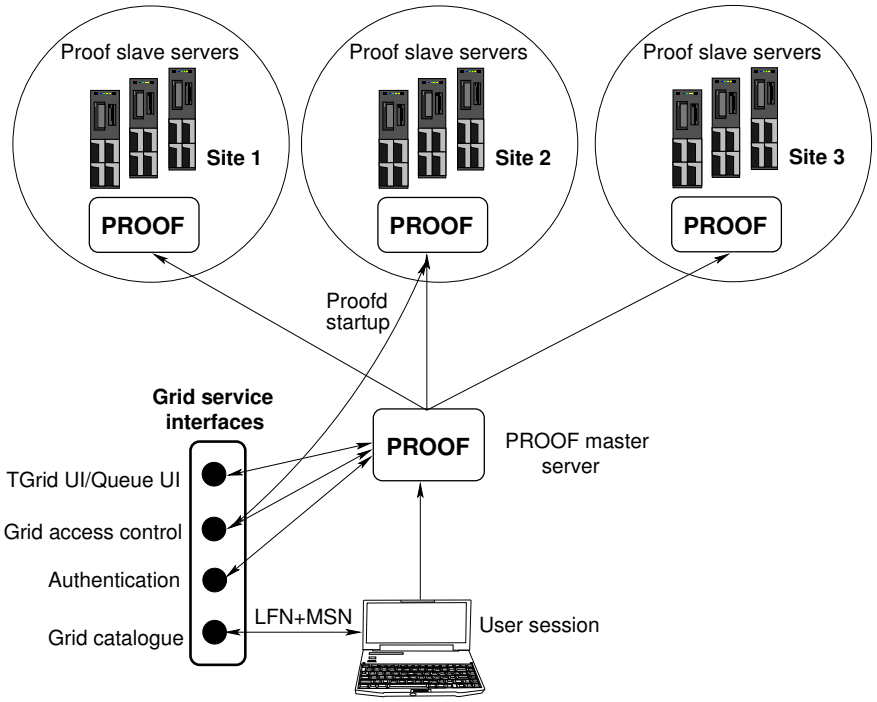

Figure 8: Setup and interaction with the Grid middleware of a user PROOF session distributed over many computing centers.

the PROOF worker servers. The user connects from the ROOT session to a master server on a remote cluster and the master server creates sub-masters and worker servers on all the nodes in the cluster. All workers process queries in parallel and the results are presented to the user as coming from a single server.

PROOF can be run either in a purely interactive way, with the user remaining connected to the master and worker servers and the analysis results being returned to the user's ROOT session for further analysis, or in an 'interactive batch' way where the user disconnects from the master and workers (see Fig. **??**). By reconnecting later to the master server the user can retrieve the analysis results for that particular query. This last mode is useful for relatively long running queries (several hours) or for submitting many queries at the same time. Both modes will be important for the analysis of ALICE data.

5.4 Analysis tools

This section is devoted to the existing analysis tools in ROOT and AliRoot. As discussed in the introduction, some very broad analysis tasks include the search for some rare events (in this case the physicist tries to maximize the signal-over-

background ratio), or measurements where it is important to maximize the signal significance. The tools that provide possibilities to apply certain selection criteria and to find the interesting combinations within a given event are described below. Some of them are very general and are used in many different places, for example the statistical tools. Others are specific to a given analysis.

5.4.1 Statistical tools

Several commonly used statistical tools are available in ROOT [**?**]. ROOT provides classes for efficient data storage and access, such as trees and ntuples. The ESD information is organized in a tree, where each event is a separate entry. This allows a chain of the ESD files to be made and the elaborated selector mechanisms to be used in order to exploit the PROOF services. Inside each ESD object the data is stored in polymorphic containers filled with reconstructed tracks, neutral particles, etc. The tree classes permit easy navigation, selection, browsing, and visualization of the data in the branches.

ROOT also provides histogramming and fitting classes, which are used for the representation of all the one- and multi-dimensional distributions, and for extraction of their fitted parameters. ROOT provides an interface to powerful and robust minimization packages, which can be used directly during some special parts of the analysis. A special fitting class allows one to decompose an experimental histogram as a superposition of source histograms.

ROOT also has a set of sophisticated statistical analysis tools such as principal component analysis, robust estimator, and neural networks. The calculation of confidence levels is provided as well.

Additional statistical functions are included in `TMath`.

5.4.2 Calculations of kinematics variables

The main ROOT physics classes include 3-vectors and Lorentz vectors, and operations such as translation, rotation, and boost. The calculations of kinematics variables such as transverse and longitudinal momentum, rapidity, pseudorapidity, effective mass, and many others are provided as well.

5.4.3 Geometrical calculations

There are several classes which can be used for measurement of the primary vertex: `AliITSVertexerZ`, `AliITSVertexerIons`, `AliITSVertexerTracks`, etc. A fast estimation of the z-position can be done by `AliITSVertexerZ`, which works for both lead–lead and proton–proton collisions. An universal tool is provided by `AliITSVertexerTracks`, which calculates the position and covariance

matrix of the primary vertex based on a set of tracks, and also estimates the χ^2 contribution of each track. An iterative procedure can be used to remove the secondary tracks and improve the precision.

Track propagation to the primary vertex (inward) is provided in AliESDtrack.

The secondary vertex reconstruction in case of V^0 is provided by `AliV0vertexer`, and in case of cascade hyperons by `AliCascadeVertexer`. An universal tool is `AliITSVertexerTracks`, which can be used also to find secondary vertexes close to the primary one, for example decays of open charm like $\mathrm{D}^0 \to \mathrm{K}^-\pi^+$ or $\mathrm{D}^+ \to \mathrm{K}^-\pi^+\pi^+$. All the vertex reconstruction classes also calculate distance of closest approach (DCA) between the track and the vertex.

The calculation of impact parameters with respect to the primary vertex is done during the reconstruction and the information is available in `AliESDtrack`. It is then possible to recalculate the impact parameter during the ESD analysis, after an improved determination of the primary vertex position using reconstructed ESD tracks.

5.4.4 Global event characteristics

The impact parameter of the interaction and the number of participants are estimated from the energy measurements in the ZDC. In addition, the information from the FMD, PMD, and T0 detectors is available. It gives a valuable estimate of the event multiplicity at high rapidities and permits global event characterization. Together with the ZDC information it improves the determination of the impact parameter, number of participants, and number of binary collisions.

The event plane orientation is calculated by the `AliFlowAnalysis` class.

5.4.5 Comparison between reconstructed and simulated parameters

The comparison between the reconstructed and simulated parameters is an important part of the analysis. It is the only way to estimate the precision of the reconstruction. Several example macros exist in AliRoot and can be used for this purpose: `AliTPCComparison.C`, `AliITSComparisonV2.C`, etc. As a first step in each of these macros the list of so-called 'good tracks' is built. The definition of a good track is explained in detail in the ITS[**?**] and TPC[**?**] Technical Design Reports. The essential point is that the track goes through the detector and can be reconstructed. Using the 'good tracks' one then estimates the efficiency of the reconstruction and the resolution.

Another example is specific to the MUON arm: the `MUONRecoCheck.C` macro compares the reconstructed muon tracks with the simulated ones.

There is also the possibility to calculate directly the resolutions without additional requirements on the initial track. One can use the so-called track label

and retrieve the corresponding simulated particle directly from the particle stack (`AliStack`).

5.4.6 Event mixing

One particular analysis approach in heavy-ion physics is the estimation of the combinatorial background using event mixing. Part of the information (for example the positive tracks) is taken from one event, another part (for example the negative tracks) is taken from a different, but 'similar' event. The event 'similarity' is very important, because only in this case the combinations produced from different events represent the combinatorial background. Typically 'similar' in the example above means with the same multiplicity of negative tracks. One may require in addition similar impact parameters of the interactions, rotation of the tracks of the second event to adjust the event plane, etc. The possibility for event mixing is provided in AliRoot by the fact that the ESD is stored in trees and one can chain and access simultaneously many ESD objects. Then the first pass would be to order the events according to the desired criterion of 'similarity' and to use the obtained index for accessing the 'similar' events in the embedded analysis loops. An example of event mixing is shown in Fig. **??**. The background distribution has been obtained using 'mixed events'. The signal distribution has been taken directly from the Monte Carlo simulation. The 'experimental distribution' has been produced by the analysis macro and decomposed as a superposition of the signal and background histograms.

5.4.7 Analysis of the High-Level Trigger (HLT) data

This is a specific analysis which is needed in order to adjust the cuts in the HLT code, or to estimate the HLT efficiency and resolution. AliRoot provides a transparent way of doing such an analysis, since the HLT information is stored in the form of ESD objects in a parallel tree. This also helps in the monitoring and visualization of the results of the HLT algorithms.

5.4.8 Visualization

The visualization classes give the possibility for prompt inspection of the simulation and reconstruction results. The initial version of the visualization is available in the `AliDisplay` class. Another more elaborated module `DISPLAY` is under development.

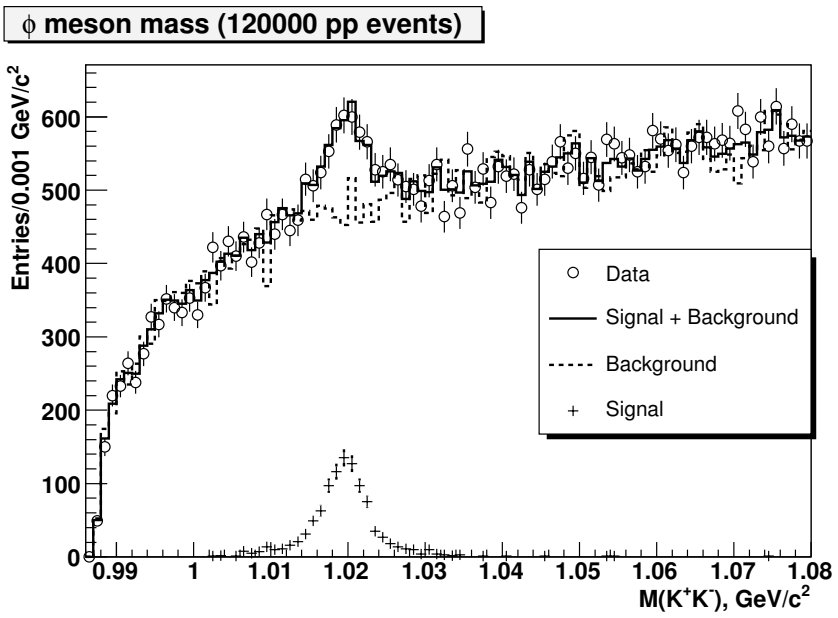

Figure 9: Mass spectrum of the ϕ meson candidates produced inclusively in the proton–proton interactions.

5.5 Existing analysis examples in AliRoot

There are several dedicated analysis tools available in AliRoot. Their results were used in the Physics Performance Report and described in ALICE internal notes. There are two main classes of analysis: the first one based directly on ESD, and the second one extracting first AOD, and then analyzing it.

- **ESD analysis**

  V^0 and cascade reconstruction/analysis

  The V^0 candidates are reconstructed during the combined barrel tracking and stored in the ESD object. The following criteria are used for the selection: minimal-allowed impact parameter (in the transverse plane) for each track; maximal-allowed DCA between the two tracks; maximal-allowed cosine of the V^0 pointing angle (angle between the momentum vector of the particle combination and the line connecting the production and decay vertexes); minimal and maximal radius of the fiducial volume; maximal-allowed χ^2. The last criterion requires the covariance matrix of track parameters, which is available only in `AliESDtrack`. The reconstruction is performed by `AliV0vertexer`. This class can be used also in the analysis. An example of reconstructed kaons taken directly from the ESDs is shown in Fig.**??**.

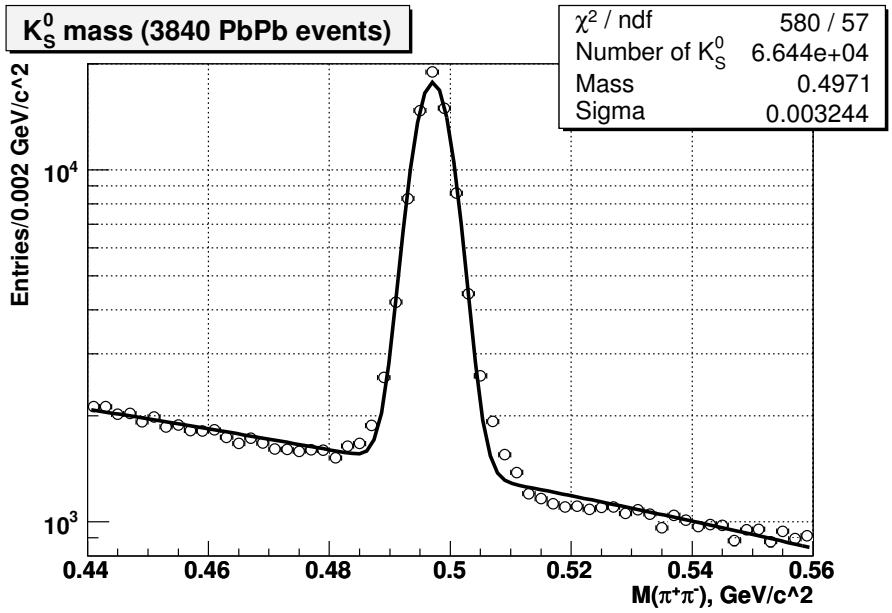

Figure 10: Mass spectrum of the K^0_S meson candidates produced inclusively in the Pb–Pb collisions.

The cascade hyperons are reconstructed using the V^0 candidate and 'bachelor' track selected according to the cuts above. In addition, one requires that the reconstructed V^0 effective mass belongs to a certain interval centered in the true value. The reconstruction is performed by `AliCascadeVertexer`, and this class can be used in the analysis.

Open charm

This is the second elaborated example of ESD analysis. There are two classes, `AliD0toKpi` and `AliD0toKpiAnalysis`, which contain the corresponding analysis code. The decay under investigation is $\mathrm{D}^0 \rightarrow \mathrm{K}^-\pi^+$ and its charge conjugate. Each D^0 candidate is formed by a positive and a negative track, selected to fulfill the following requirements: minimal-allowed track transverse momentum, minimal-allowed track impact parameter in the transverse plane with respect to the primary vertex. The selection criteria for each combination include maximal-allowed distance of closest approach between the two tracks, decay angle of the kaon in the D^0 rest frame in a given region, product of the impact parameters of the two tracks larger than a given value, pointing angle between the D^0 momentum and flight-line smaller than a given value. The particle identification probabilities are used to reject the wrong combinations, namely (K, K) and (π, π), and to enhance the signal-to-background ratio at low momentum by requiring the kaon identification. All proton-tagged tracks are excluded before the analysis loop on track pairs. More details can be found in Ref.[**?**].

Quarkonia analysis

Muon tracks stored in the ESD can be analyzed for example by the macro `MUONmassPlot_ESD.C`. This macro performs an invariant-mass analysis of muon unlike-sign pairs and calculates the combinatorial background. Quarkonia $p_{\mathrm{t}}$and rapidity distribution are built for J/ψ and Υ. This macro also performs a fast single-muon analysis: p_{t}, rapidity, and θ vs φ acceptance distributions for positive and negative muon tracks with a maximal-allowed χ^2.

- **AOD analysis**

  Often only a small subset of information contained in the ESD is needed to perform an analysis. This information can be extracted and stored in the AOD format in order to reduce the computing resources needed for the analysis.

  The AOD analysis framework implements a set of tools like data readers, converters, cuts, and other utility classes. The design is based on two main requirements: flexibility and common AOD particle interface. This guarantees that several analyses can be done in sequence within the same computing session.

  In order to fulfill the first requirement, the analysis is driven by the 'analysis manager' class and particular analyses are added to it. It performs the loop

over events, which are delivered by an user-specified reader. This design allows the analyses to be ordered appropriately if some of them depend on the results of the others.

The cuts are designed to provide high flexibility and performance. A two-level architecture has been adopted for all the cuts (particle, pair and event). A class representing a cut has a list of 'base cuts'. Each base cut implements a cut on a single property or performs a logical operation (and, or) on the result of other base cuts.

A class representing a pair of particles buffers all the results, so they can be re-used if required.

Particle momentum correlations (HBT) – HBTAN module

Particle momentum correlation analysis is based on the event-mixing technique. It allows one to extract the signal by dividing the appropriate particle spectra coming from the original events by those from the mixed events.

Two analysis objects are currently implemented to perform the mixing: the standard one and the one implementing the Stavinsky algorithm[**?**]. Others can easily be added if needed.

An extensive hierarchy of the function base classes has been implemented facilitating the creation of new functions. A wide set of the correlation, distribution and monitoring functions is already available in the module. See Ref.[**?**] for the details.

The package contains two implementations of weighting algorithms, used for correlation simulations (the first developed by Lednicky [**?**] and the second due to CRAB [**?**]), both based on an uniform interface.

Jet analysis

The jet analysis[**?**] is available in the module JETAN. It has a set of readers of the form `AliJetParticlesReader<XXX>`, where `XXX = ESD`, `HLT`, `KineGoodTPC`, `Kine`, derived from the base class `AliJetParticlesReader`. These provide an uniform interface to the information from the kinematics tree, from HLT, and from the ESD. The first step in the analysis is the creation of an AOD object: a tree containing objects of type `AliJetEventParticles`. The particles are selected using a cut on the minimal-allowed transverse momentum. The second analysis step consists of jet finding. Several algorithms are available in the classes of the type `Ali<XXX>JetFinder`. An example of AOD creation is provided in the `createEvents.C` macro. The usage of jet finders is illustrated in `findJets.C` macro.

V^0 AODs

The AODs for V^0 analysis contain several additional parameters, calculated and stored for fast access. The methods of the class `AliAODv0` provide access to all the geometrical and kinematics parameters of a V^0 candidate, and to the ESD information used for the calculations.

MUON

There is also a prototype MUON analysis provided in `AliMuonAnalysis`. It simply fills several histograms, namely the transverse momentum and rapidity for positive and negative muons, the invariant mass of the muon pair, etc.

The analysis framework is one of the most active fields of development. Many new classes and macros are in preparation. Some of them are already tested on the data produced during the Physics Data Challenge 2004 and will become part of the ALICE software.

6 Analysis Foundation Library

The result of the reconstruction chain is the Event Summary Data (ESD) object. It contains all the information that may be useful in *any* analysis. In most cases only a small subset of this information is needed for a given analysis. Hence, it is essential to provide a framework for analyses, where user can extract only the information required and store it in the Analysis Object Data (AOD) format. This is to be used in all his further analyses. The proper data preselecting allows to speed up the computation time significantly. Moreover, the interface of the ESD classes is designed to fulfill the requirements of the reconstruction code. It is inconvenient for most of analysis algorithms, in contrary to the AOD one. Additionally, the latter one can be customized to the needs of particular analysis, if it is only required.

We have developed the analysis foundation library that provides a skeleton framework for analyses, defines AOD data format and implements a wide set of basic utility classes which facilitate the creation of individual analyses. It contains classes that define the following entities:

- AOD event format
- Event buffer
- Particle(s)
- Pair

- Analysis manager class
- Base class for analyses
- Readers
- AOD writer
- Particle cuts
- Pair cuts
- Event cuts
- Other utility classes

It is designed to fulfill two main requirements:

1. **Allows for flexibility in designing individual analyses** Each analysis has its most performing solutions. The most trivial example is the internal representation of a particle momentum: in some cases the Cartesian coordinate system is preferable and in other cases - the cylindrical one.
2. **All analyses use the same AOD particle interface to access the data** This guarantees that analyses can be chained. It is important when one analysis depends on the result of the other one, so the latter one can process exactly the same data without the necessity of any conversion. It also lets to carry out many analyses in the same job and consequently, the computation time connected with the data reading, job submission, etc. can be significantly reduced.

The design of the framework is described in detail below.

6.1 AOD

The `AliAOD` class contains only the information required for an analysis. It is not only the data format as they are stored in files, but it is also used internally throughout the package as a particle container. Currently it contains a `TClonesArray` of particles and data members describing the global event properties. This class is expected to evolve further as new analyses continue to be developed and their requirements are implemented.

6.2 Particle

`AliVAODParticle` is a pure virtual class that defines a particle interface. Each analysis is allowed to create its own particle class if none of the already existing ones meet its requirements. Of course, it must derive from `AliVAODParticle`. However, all analyses are obliged to use the interface defined in `AliVAODParticle` exclusively. If additional functionality is required, an appropriate method is also added to the virtual interface (as a pure virtual or an empty one). Hence, all other analyses can be ran on any AOD, although the processing time might be longer in some cases (if the internal representation is not the optimal one).

We have implemented the standard concrete particle class called `AliAODParticle`. The momentum is stored in the Cartesian coordinates and it also has the data members describing the production vertex. All the PID information is stored in two dynamic arrays. The first array contains probabilities sorted in descending order, and the second one - corresponding PDG codes (Particle Data Group). The PID of a particle is defined by the data member which is the index in the arrays. This solution allows for faster information access during analysis and minimizes memory and disk space consumption.

6.3 Pair

The pair object points to two particles and implements a set of methods for the calculation of the pair properties. It buffers calculated values and intermediate results for performance reasons. This solution applies to quantities whose computation is time consuming and also to quantities with a high reuse probability. A Boolean flag is used to mark the variables already calculated. To ensure that this mechanism works properly, the pair always uses its own methods internally, instead of accessing its variables directly.

The pair object has pointer to another pair with the swapped particles. The existence of this feature is connected to the implementation of the mixing algorithm in the correlation analysis package: if particle A is combined with B, the pair with the swapped particles is not mixed. In non-identical particle analysis their order is important, and a pair cut may reject a pair while a reversed one would be accepted. Hence, in the analysis the swapped pair is also tried if a regular one is rejected. In this way the buffering feature is automatically used also for the swapped pair.

6.4 Analysis manager class and base class

The *analysis manager* class (`AliRunAnalysis`) drives all the process. A particular analysis, which must inherit from `AliAnalysis` class, is added to it. The user triggers analysis by calling the `Process` method. The manager performs a loop

over events, which are delivered by a reader (derivative of the `AliReader` class, see section **??**). This design allows to chain the analyses in the proper order if any depends on the results of the other one.

The user can set an event cut in the manager class. If an event is not rejected, the `ProcessEvent` method is executed for each analysis object. This method requires two parameters, namely pointers to a reconstructed and a simulated event.

The events have a parallel structure, i.e. the corresponding reconstructed particles and simulated particles have always the same index. This allows for easy implementation of an analysis where both are required, e.g. when constructing residual distributions. It is also very important in correlation simulations that use the weight algorithm[**?**]. By default, the pointer to the simulated event is null, i.e. like it is in the experimental data processing.

An event cut and a pair cut can be set in `AliAnalysis`. The latter one points two particle cuts, so an additional particle cut data member is redundant because the user can set it in this pair cut.

`AliAnalysis` class has the feature that allows to choose which data the cuts check:

1. the reconstructed (default)
2. the simulated
3. both.

It has four pointers to the method (data members):

1. `fkPass1` – checks a particle, the cut is defined by the cut on the first particle in the pair cut data member
2. `fkPass2` – as above, but the cut on the second particle is used
3. `fkPass` – checks a pair
4. `fkPassPairProp` – checks a pair, but only two particle properties are considered

Each of them has two parameters, namely pointers to reconstructed and simulated particles or pairs. The user switches the behavior with the method that sets the above pointers to the appropriate methods. We have decided to implement this solution because it performs faster than the simpler one that uses boolean flags and ”if” statements. These cuts are used mostly inside multiply nested loops, and even a small performance gain transforms into a noticeable reduction of the overall computation time. In the case of an event cut, the simpler solution was applied. The `Rejected` method is always used to check events. A developer of the analysis code must always use this method and the pointers to methods itemized above to benefit from this feature.

6.5 Readers

A Reader is the object that provides data far an analysis. `AliReader` is the base class that defines a pure virtual interface.

A reader may stream the reconstructed and/or the simulated data. Each of them is stored in a separate AOD. If it reads both, a corresponding reconstructed and simulated particle have always the same index.

Most important methods for the user are the following:

- `Next` – It triggers reading of a next event. It returns 0 in case of success and 1 if no more events are available.
- `Rewind` – Rewinds reading to the beginning
- `GetEventRec` and `GetEventSim` – They return pointers to the reconstructed and the simulated events respectively.

The base reader class implements functionality for particle filtering at a reading level. A user can set any number of particle cuts in a reader and the particle is read if it fulfills the criteria defined by any of them. Particularly, a particle type is never certain and the readers are constructed in the way that all the PID hypotheses (with non-zero probability) are verified. In principle, a track can be read with more than one mass assumption. For example, consider a track which in 55% is a pion and in 40% a kaon, and a user wants to read all the pions and kaons with the PID probabilities higher then 50% and 30%, respectively. In such cases two particles with different PIDs are added to AOD. However, both particle have the same Unique Identification number (UID) so it can be easily checked that in fact they are the same track.

`AliReader` implements the feature that allows to specify and manipulate multiple data sources, which are read sequentially. The user can provide a list of directory names where the data are searched. The `ReadEventsFromTo` method allows to limit the range of events that are read (e.g. when only one event of hundreds stored in an AOD is of interest). `AliReader` has the switch that enables event buffering, so an event is not deleted and can be quickly accessed if requested again.

Particles within an event are frequently sorted in some way, e.g. the particle trajectory reconstruction provides tracks sorted according to their transverse momentum. This leads to asymmetric distributions where they are not expected. The user can request the reader to randomize the particle order with `SetBlend` method.

The AOD objects can be written to disk with the `AliReaderAOD` using the static method `WriteAOD`. As the first parameter user must pass the pointer to another reader that provides AOD objects. Typically it is `AliReaderESD`, but it also

can be other one, f.g. another AliReaderAOD (to filter out the desired particles from the already existing AODs).

Inside the file, the AODs are stored in a TTree. Since the AOD stores particles in the clones array, and many particles formats are allowed, the reading and writing is not straight forward. The user must specify what is the particle format to be stored on disk, because in a general case the input reader can stream AODs with not consistent particle formats. Hence, the careful check must be done, because storing an object of the different type then it was specified in the tree leads to the inevitable crash. If the input AOD has the different particle type then expected it is automatically converted. Hence, this method can be also used for the AOD type conversion.

6.6 AODs buffer

Normally the readers do not buffer the events. Frequently an event is needed to be kept for further analysis, f.g. if uncorrelated combinatorial background is computed. We have implemented the FIFO (First In First Out) type buffer called AliEventBuffer that caches the defined number of events.

6.7 Cuts

The cuts are designed to guarantee the highest flexibility and performance. We have implemented the same two level architecture for all the cuts (particle, pair and event). Cut object defines the ranges of many properties that a particle, a pair or an event may posses and it also defines a method, which performs the necessary check. However, usually a user wants to limit ranges of only a few properties. For speed and robustness reasons, the design presented in Fig.**??** was developed.

The cut object has an array of pointers to base cuts. The number of entries in the array depends on the number of the properties the user wants to limit. The base cut implements checks on a single property. It implements maximum and minimum values and a virtual method Rejected that performs a range check of the value returned by pure virtual method GetValue. Implementation of a concrete base cut is very easy in most cases: it is enough to implement GetValue method. The ANALYSIS package already implements a wide range of base cuts, and the cut classes have a comfortable interface for setting all of them. For example it is enough to invoke the SetPtRange(min,max) method and behind the scenes a proper base cut is created and configured.

The base cuts performing a logical operation (and,or) on the result of two other base cuts are also implemented. This way the user can configure basically any cut in a macro. Supplementary user defined base cuts can be added in the user

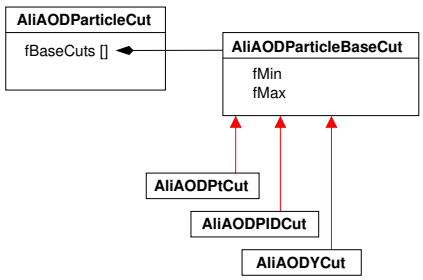

Figure 11: Cut classes diagram on the example of the particle cut.

provided libraries. In case the user prefers to implement a complicated cut in a single method (class) he can create his base cut performing all the operations.

The pair cut in addition to an array of pointers to the base pair cuts it has two pointers to particle cut, one for each particle in the pair.

6.8 Other classes

We have developed a few classes that are used in correlation analyses, but they can be also useful in the others. The first is the TPC cluster map, which is the bitmap vector describing at which pad-rows a track has a cluster. It is used by the anti-splitting algorithm in the particle correlation analysis.

Another example is the `AliTrackPoints` class, that stores track space coordinates at requested distances from the center of the detector. It is used in the particle correlation analysis by the anti-merging cut. The coordinates are calculated assuming the helix shape of a track. Different options that define the way they are computed are available.

7 Data input, output and exchange subsystem of AliRoot

This section is taken from[**?**].

A few tens of different data types is present within AliRoot because hits, summable digits, digits and clusters are characteristic for each sub-detector. Writing all of the event data to a single file, as it used to be implemented, was causing number of limitations. Moreover, the reconstruction chain introduces rather complicated dependences between different components of the framework, what is highly undesirable from the point of view of software design. In order to solve both problems, we have designed a set of classes that manage data manipulation

i.e. storage, retrieval and exchange within the framework.

It was decided to use the "white board" concept, which is a single exchange object where were all data are stored and publicly accessible. For that purpose I have employed **TFolder** facility of ROOT. This solution solves the problem of inter-module dependencies.

There are two most frequently occurring use-cases concerning the way a user deals with the data within the framework:

1. data production – produce - **write** - **unload** (clean)
2. data processing – **load** (retrieve) - process - **unload**

Loaders are the utility classes that encapsulate and automatize the tasks written in bold font. They limit the user's interaction with the I/O routines to the necessary minimum, providing friendly and very easy interface, which for the use-cases considered above, consists of only 3 methods:

- `Load` – retrieves the requested data to the appropriate place in the white board (folder)
- `Unload` – cleans the data
- `Write` – writes the data

Such an insulation layer has number of advantages. It

- makes the data access easier for the user.
- allows to avoid the code duplication throughout the framework.
- minimize the risk of a bug occurrence resulting from the improper I/O management. The ROOT object oriented data storage extremely simplifies the user interface, however, there are a few pitfalls that are frequently unknown to an unexperienced user.

To make the description more clear we need to introduce briefly basic concepts and the way the AliRoot program operates. The basic entity is an event, i.e. all the data recorded by the detector in a certain time interval plus all the reconstructed information from these data. Ideally the data are produced by the single collision selected by a trigger for recording. However, it may happen that the data from the previous or proceeding events are present because the bunch crossing rate is higher then the maximum detector frequency (pile-up), or simply more than one collision occurred within one bunch crossing.

Information describing the event and the detector state is also stored, like bunch crossing number, magnetic field, configuration, alignment, etc.., In the case

of a Monte-Carlo simulated data, information concerning the generator, simulation parameters is also kept. Altogether this data is called the **header**.

For the collisions that produce only a few tracks (best example are the pp collisions) it may happen that total overhead (the size of the header and the ROOT structures supporting object oriented data storage) is non-negligible in comparison with the data itself. To avoid such situations, the possibility of storing an arbitrary number of events together within a **run** is required. Hence, the common data can be written only ones per run and several events can be written to a single file.

It was decided that the data related to different detectors and different processing phases should be stored in different files. In such a case only the required data need to be downloaded for an analysis. It also allows to alter the files easily if required, for example when a new version of the reconstruction or simulation is needed to be run for a given detector. Hence, only new files are updated and all the rest may stay untouched. It is especially important because mass storage systems practically do not allow to erase files. This also gives the possibility for an easy comparison of the data produced with competing algorithms.

All the header data, configuration and management objects are stored in a separate file, which is usually named galice.root (for simplicity we will further refer to it as galice).

7.1 The "White Board"

The folder structure is presented in Fig.**??**. It is subdivided into two parts:

- **event data** that have the scope of single event
- **static data** that do not change from event to event, i.e. geometry and alignment, calibration, etc.

During startup of AliRoot the skeleton structure of the ALICE white board is created. The `AliConfig` class (singleton) provides all the functionality that is needed to construct the folder structures.

An event data are stored under a single sub-folder (event folder) named as specified by the user when opening a session (run). Many sessions can be opened at the same time, providing that each of them has an unique event folder name, so they can be distinguished by this name. This functionality is crucial for superimposing events on the level of the summable digits, i.e. analog detector response without the noise contribution (the event merging). It is also useful when two events or the same event simulated or reconstructed with a competing algorithm, need to be compared.

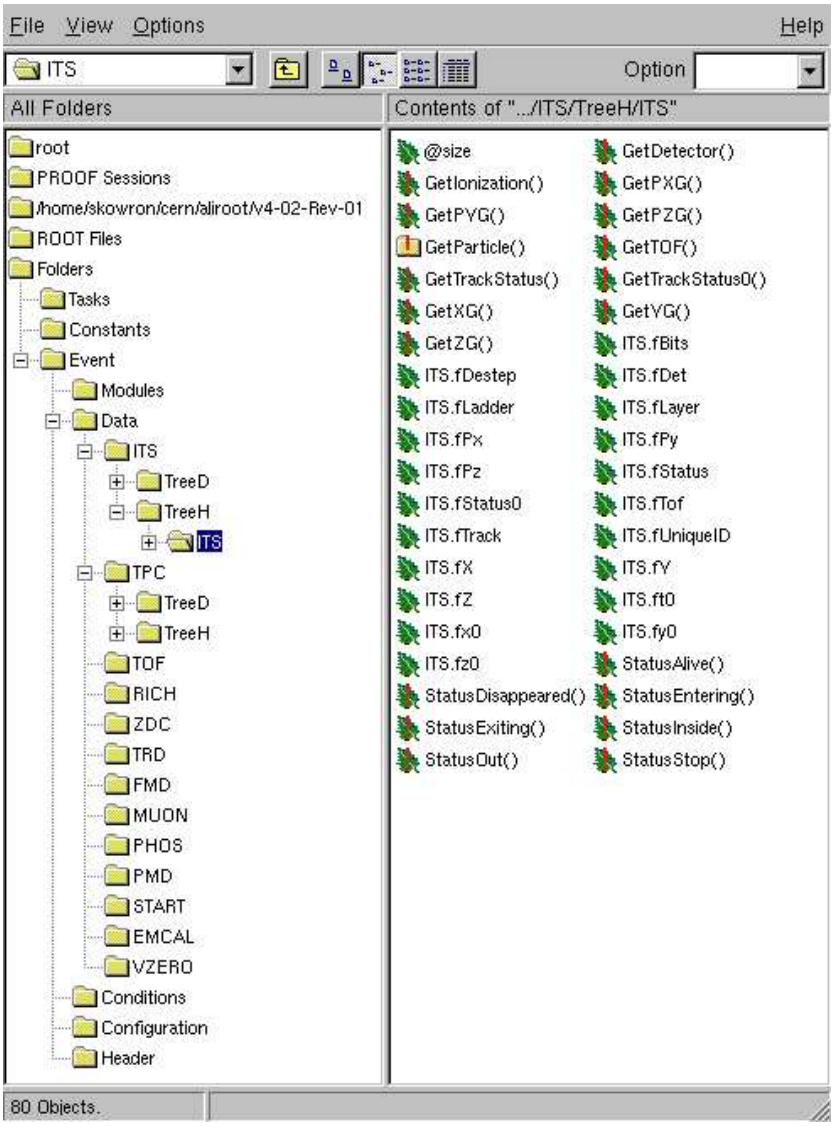

Figure 12: The folders structure. An example event is mounted under “Event” folder.

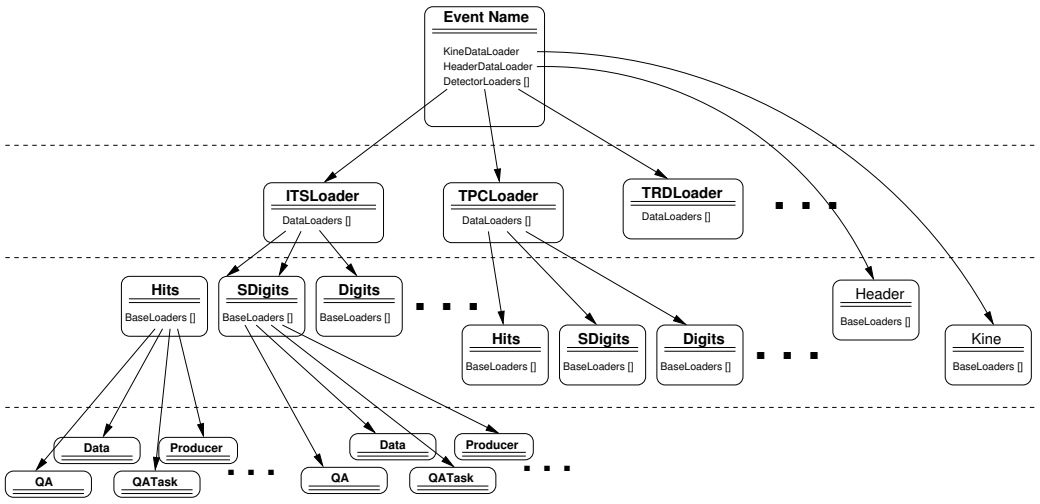

Figure 13: Loaders diagram. Dashed lines separate layers serviced by the different types of the loaders (from top): AliRunLoder, AliLoader, AliDataLoader, AliBaseLoader.

7.2 Loaders

I have implemented a set of classes that simplify the data access and storage. The loaders can be represented as a four layer, tree like structure (see Fig.**??**). It represents the logical structure of the detector and the data association.

1. `AliBaseLoader` – One base loader is responsible for posting (finding in a file and putting in a folder) and writing (finding in a folder and putting in a file) of a single object. AliBaseLoader is a pure virtual class because writing and posting depend on the type of an object. the following concrete classes are currently implemented:

   - `AliObjectLoader` – It handles `TObject`, i.e. basically any object within ROOT and AliRoot since an object must inherit from this class to be posted to the white board (added to `TFolder`).
   - `AliTreeLoader` – It is the base loader for `TTrees`, which requires special handling, because they must be always properly associated with a file.
   - `AliTaskLoader` – It handles `TTask`, which need to be posted to the appropriate parental `TTask` instead of `TFolder`.

   `AliBaseLoader` has always the same name as the object it manages (to be able to find it in a file or folder). The user normally does not need to

use these classes directly and they are rather utility classes employed by `AliDataLoader`.

2. `AliDataLoader` – It manages a single data type, for example digits for a detector or kinematics tree. Since a few objects are normally associated with a given data type (data itself, quality assurance data (QA), a task that produces the data, QA task, etc.) `AliDataLoader` has an array of `AliBaseLoaders`, so each of them is responsible for each object. Hence, `AliDataLoader` can be configured individually to meet specific requirements of a certain data type.

   A single file contains the data corresponding to a single processing phase and solely of one detector. By default the file is named according to the schema *Detector Name* + *Data Name* + *.root* but it can be changed in runtime if needed so the data can be stored in or retrieved from an alternative source. When needed, the user can limit the number of events stored in a single file. If the maximum number is exceeded, a file is closed and a new one is opened with the consecutive number added to its name before *.root* suffix. Of course, during the reading process, files are also automatically interchanged behind the scenes and it is invisible to the user.

   The `AliDataLoader` class performs all the tasks related to file management e.g. opening, closing, ROOT directories management, etc. Hence, for each data type the average file size can be tuned. It is important because it is undesirable to store small files on the mass storage systems and on the other hand, all file systems have a maximum file size allowed (e.g. currently 2 GB on the Linux ext2).

3. `AliLoader` – It manages all the data associated with a single detector (hits, digits, summable digits, reconstructed points, etc.). It has an array of `AliDataLoaders` and each of them manages a single data type.

   The `AliLoader` object is created by an class representing a detector (inheriting from `AliDetector`). Its functionality can be extended and customized to the needs of a particular detector by creating a specialized class that derives from `AliLoader`, as it was done, for instance, for ITS or PHOS. The default configuration can be easily modified either in `AliDetector::MakeLoader` or by overriding the method `AliLoader::InitDefaults`.

4. `AliRunLoader` – It is a main handle for data access and manipulation in AliRoot. There is only one such an object in each run. It is always named *RunLoader* and stored on the top (ROOT) directory of a galice file.

   It keeps an array of `AliLoader`'s, one for each detector. It also manages

the event data that are not associated with any detector i.e. Kinematics and Header and it utilizes `AliDataLoader`'s for this purpose.

The user opens a session using a static method `AliRunLoader::Open`. This method has three parameters: the file name, event folder name and mode. The mode can be "new" and in this case a file and a run loader are created from scratch. Otherwise, a file is opened and a run loader is searched in. If successful, the event folder with a provided name (if such does not exist yet) is created and the structure presented in Fig.**??** is created within the folder. The run loader is put in the event folder, so the user can always find it there and use it for data management.

`AliRunLoader` provides a simple method `GetEvent(n)` to loop over events within a run. Calling it causes that all currently loaded data are cleaned and the data for the newly requested event are automatically posted.

In order to facilitate the way the user interacts with the loaders, `AliRunLoader` provides the wide set of shortcut methods. For example, if digits are required to be loaded, the user can call `AliRunLoader::LoadDigits("ITS TPC")`, instead of finding the appropriate `AliDataLoader`'s responsible for digits for ITS and TPC, and then request to load the data for each of them.

8 Calibration and alignment

8.1 Calibration framework

The calibration framework is based on the following principles:

- the calibration and alignment database contains ROOT TObjects stored into ROOT files;
- calibration and alignment objects are RUN DEPENDENT objects;
- the database is READ-ONLY (automatic versioning of the stored objects)
- three different database structures are (currently) available:
  - a GRID folder containing Root files, each one containing one single Root object. The Root files are created inside a directory tree defined by the object's name and run validity range;
  - a LOCAL folder containing Root files, each one containing one single Root object, with a structure similar to the Grid one;

  - a LOCAL Root (dump) file containing one or more objects. The objects are stored into Root TDirectories defined by the object's name and run range.

- object storing and retrieval techniques are transparent to the user: he/she should only specify the kind of storage he wants to use (”grid”, ”local”, ”dump”). Object are stored and retrieved using the AliCDBStorage public classes:

```
Bool_t AliCDBStorage::Put(...)
    and
AliCDBEntry* AliCDBStorage::Get(...)
```

  In addition, multiple objects can be retrieved using:

```
TList* AliCDBStorage::GetAll(...) (returns list of AliCDBEntry objects).
```

- During object retrieval, the user has the possibility to retrieve the highest version of the object or to specify a particular version by means of one or more selection criteria.

New features of the CDB storage classes

A schema of the new framework can be found on the slides of the September 1 Alice weekly meeting talk (page 6). The main new features are:

- new MANAGER class AliCDBManager. It is a singleton which handles the instantiation, usage and destruction of all the storage classes. It allows the instantiation of more than one storage type at a time, keeping tracks of the list of active storages. The instantiation of a storage element is done by means of AliCDBManager public method GetStorage. A storage element is identified by its ”URI” (a string) or by its ”parameters”. The set of parameters defining each storage is contained in its specific AliCDBParam class (AliCDBGridParam, AliCDBLocalParam, AliCDBDumpParam).

- new versioning schema. In order to avoid version clashes when objects are transferred from grid to local and vice versa, we have introduced a new versioning schema. Basically the objects are defined by TWO version numbers: a ”Grid” version and a ”Local” version (subVersion). In local storage

only the local version is increased, while in Grid storage only the Grid version is increased. When the object is transferred from local to Grid the Grid version is increased by one; when the object is transferred from Grid to Local the Grid version is kept and the subVersion is reset to zero. You can find a plot of this schema on my talk (page 11).

- The container class of the object and its metadata (AliCDBEntry) has been redesigned. The metadata of the object has been divided into two classes: one which contains the data used to store and retrieve the object (”identity” of the object, AliCDBId) and the other containing the metadata which is not used during storage and retrieval (AliCDBMetaData).

  The AliCDBId object in turn contains:

  - an object describing the name (path) of the object (AliCDBPath). The path name must have a fixed, three-level directory structure: ”level1/level2/level3”
  - an object describing the run validity range of the object (AliCDBRunRange)
  - the version and subversion numbers (automatically set during storage)
  - a string (fLastStorage) specifying from which storage the object was retrieved (”new”, ”grid”, ”local”, ”dump”)

  The AliCDBId object has two functions:

  - during storage it is used to specify the path and run range of the object;
  - during retrieval it is used as a ”query”: it contains the path of the object, the required run and (if needed) the version and subversion to be retrieved (if version and/or subversion are not specified the highest ones are looked for).

Some usage examples

The following use cases are illustrated:

- A pointer to the single instance of the AliCDBManager class is obtained with:

  ```
  AliCDBManager::Instance()
  ```

- A storage is activated and a pointer to it is returned using the `AliCDBManager::GetStorage(const char* URI)` method. Here are some examples of how to activate a storage via an URI string. The URI’s must have a well defined syntax, for example (local cases):

  - "local://DBFolder" to local storage with base directory "DBFolder" created (if not existing from the working directory)
  - "local://$ALICE_ROOT/DBFolder" to local storage with base directory "$ALICE_ROOT/DBFolder" (full path name)
  - "dump://DBFile.root" to Dump storage. The file DBFile.root is looked for or created in the working directory if the full path is not specified
  - "dump://DBFile.root;ReadOnly" to Dump storage. DBFile.root is opened in read only mode.

- Concrete examples (local case):

```
AliCDBStorage *sto =
  AliCDBManager::Instance()->GetStorage("local://DBFolder"):

AliCDBStorage *dump =
  AliCDBManager::Instance()->GetStorage("dump:///data/DBFile.root;ReadOnly"):
```

- Creation and storage of an object. Example of how an object can be created and stored in a local database

  - Let's suppose our object is an AliZDCCalibData object (container of arrays of pedestals constants), whose name is "ZDC/Calib/Pedestals" and is valid for run 1 to 10.

```
AliZDCCalibData *calibda = new AliZDCCalibData();
// ... filling calib data...

// creation of the AliCDBId object (identifier of the object)
AliCDBId id("ZDC/Calib/Pedestals",1,10);

// creation and filling of the AliCDBMetaData
AliCDBMetaData *md = new AliCDBMetaData();
md->Set... // fill meta data object, see list of setters...

// Activation of local storage
AliCDBStorage *sto =
AliCDBManager::Instance()->GetStorage("local://$HOME/DBFolder");

// put object into database
```

```
sto->Put(calibda, id, md);
```

The object is stored into local file: $HOME/DBFolder/ZDC/Calib/Pedestals/Run1_10_v0_s0.roo

- Examples of how to retrieve an object

```
// Activation of local storage
AliCDBStorage *sto =
AliCDBManager::Instance()->GetStorage("local://$HOME/DBFolder");

// Get the AliCDBEntry which contains the object "ZDC/Calib/Pedestals",
valid for run 5, highest version
AliCDBEntry* entry = sto->Get("ZDC/Calib/Pedestals",5)
// alternatively, create an AliCDBId query and use sto->Get(query) ...

// specifying the version: I want version 2
AliCDBEntry* entry = sto->Get("ZDC/Calib/Pedestals",5,2)

// specifying version and subversion: I want version 2 and subVersion 1
AliCDBEntry* entry = sto->Get("ZDC/Calib/Pedestals",5,2,1)
```

- Selection criteria can be also specified using `AliCDBStorage::AddSelection(...)` methods:

```
// I want version 2\_1 for all "ZDC/Calib/*" objects for runs 1-100
sto->AddSelection("ZDC/Calib/*",1,100,2,1);
// and I want version 1\_0 for "ZDC/Calib/Pedestals" objects for runs 5-10
sto->AddSelection("ZDC/Calib/Pedestals",5,10,1,0)

AliCDBEntry* entry = sto->Get("ZDC/Calib/Pedestals",5)
```

See also: `AliCDBStorage::RemoveSelection(...)`, `RemoveAllSelections()`, `PrintSelectionList()`

- Retrieval of multiple objects with `AliCDBStorage::GetAll()`

```
TList *list = sto->GetAll("ZDC/*",5)
```

- Use of Default storage and Drain storages

AliCDBManager allows to set pointers to a ”default storage” and to a ”drain storage”. In particular, if the drain storage is set, all the retrieved objects are automatically stored into it.

The default storage is automatically set as the first active storage. To set the default storage to another storage:

```
AliCDBManager::Instance()->SetDefaultStorage("uri")
```

The default storage can be then used by:

```
AliCDBEntry *entry =
  AliCDBManager::Instance()->GetDefaultStorage()->Get(...)
```

The drain storage can be set in a similar way:

```
AliCDBManager::Instance()->SetDrain("uri")
```

There are some AliCDBManager public methods to handle the default and storage methods:

```
Bool_t  IsDefaultStorageSet()
void RemoveDefaultStorage()
Bool_t  IsDrainSet()
void RemoveDrain()
```

- Example of how to use default and drain storage:

```
AliCDBManager::Instance()->SetDefaultStorage("local://$HOME/DBFolder");
AliCDBManager::Instance()->SetDrain("dump://$HOME/DBDrain.root");

AliCDBEntry *entry =
AliCDBManager::Instance()->GetDefaultStorage()->Get("ZDC/Calib/Pedestals",5)
// Retrieved entry is automatically stored into DBDrain.root !
```

- To destroy the AliCDBManager instance and all the active storages:

```
AliCDBManager::Instance()->Destroy()
```

References

[1] CERN/LHCC 2003-049, ALICE Physics Performance Report, Volume 1 (7 November 2003);
ALICE Collaboration: F. Carminati *et al.*, J. Phys. G: Nucl. Part. Phys. **30** (2004) 1517–1763.

[2] CERN-LHCC-2005-018, ALICE Technical Design Report: Computing, ALICE TDR 012 (15 June 2005).

[3] `http://www.redhat.com`

[4] `http://fedora.redhat.com`

[5] `http://www.linux.org`

[6] `http://linux.web.cern.ch/linux`

[7] `http://gcc.gnu.org`

[8] `http://www.intel.com/cd/software/products/asmo-na/eng/compilers/index.htm`

[9] `http://www.intel.com/cd/software/products/asmo-na/eng/vtune/index.htm`

[10] `http://www.intel.com/products/processor/itanium2/index.htm`

[11] `http://www.amd.com`

[12] `http://www.cvshome.org`

[13] `http://ximbiot.com/cvs/manual`

[14] `http://root.cern.ch`

[15] `http://cewrn.ch/geant4`

[16] `http://cern.ch/clhep`

[17] `http://www.fluka.org`

[18] `http://cern.ch/castor`

[19] X. N. Wang and M. Gyulassy, Phys. Rev. **D44** (1991) 3501.
M. Gyulassy and X. N. Wang, Comput. Phys. Commun. **83** (1994) 307-331.
The code can be found in http://www-nsdth.lbl.gov/∼xnwang/hijing/

[20] J. Ranft, Phys. Rev. **D 51** (1995) 64.

[21] ALICE-INT-2003-036

[22] F. Abe *et al.*, (CDF Collaboration), Phys. Rev. Lett.**61** (1988) 1819.

[23] B. Andersson, *et al.,* Phys. Rep. **97** (1983) 31.

[24] B. Andersson, *et al.,* Nucl. Phys. **B281** (1987) 289;
B. Nilsson-Almqvist and E. Stenlund, Comput. Phys. Commun. **43** (1987) 387.

[25] A. Capella, *et al.,* Phys. Rep. **236** (1994) 227.

[26] H.-U. Bengtsson and T. Sjostrand, Comput. Phys. Commun. **46** (1987) 43;
the code can be found in `http://nimis.thep.lu.se/~torbjorn/Pythia.html`
T. Sjostrand, Comput. Phys. Commun. **82** (1994) 74;
the code can be found in http://www.thep.lu.se/~torbjorn/Pythia.html

[27] A. Morsch, `http://home.cern.ch/~morsch/AliGenerator/AliGenerator.html`
and `http://home.cern.ch/~morsch/generator.html`

[28] NA35 Collaboration, T. Alber et al., Z. Phys. **C 64** (1994) 195.

[29] NA35 Collaboration, T. Alber et al., Eur. Z. Phys. **C2** (1998) 643.

[30] D. Kharzeev: Phys. Lett. **B 378** (1996) 238.

[31] A. Capella and B. Kopeliovich, Phys. Lett. **B381** (1996) 325.

[32] R. V. Barrett and D. F. Jackson, *Nuclear sizes and structure,* Clarendon Press, Oxford, 1977.

[33] S. Roesler, R. Engel and J. Ranft, Phys. Rev. **D57** (1998) 2889.

[34] S. Roesler, personal communication, 1999.

[35] M. Glück, E. Reya and A. Vogt: Z. Phys. **C67** (1995) 433.

[36] M. Glück, E. Reya and A. Vogt, Eur. Phys. J. **C5** (1998) 461.

[37] L. Ray and R.S. Longacre, STAR Note 419.

[38] S. Radomski and Y. Foka, ALICE Internal Note 2002-31.

[39] `http://radomski.home.cern.ch/~radomski/AliMevSim.html`

[40] `http://home.cern.ch/~radomski`

[41] L. Ray and G.W. Hoffmann. Phys. Rev. **C 54**, (1996) 2582, Phys. Rev. **C60**, (1999) 014906.

[42] P. K. Skowroński, ALICE HBT Web Page, http://aliweb.cern.ch/people/skowron

[43] A. Alscher, K. Hencken, D. Trautmann, and G. Baur. Phys. Rev. A **55**, (1997) 396.

[44] K. Hencken, Y. Kharlov, and S. Sadovsky, ALICE Internal Note 2002-27.

[45] A.M. Poskanzer and S.A. Voloshin, Phys. Rev. **C 58**, (1998) 1671.

[46] http://nuclear.ucdavis.edu/ jklay/ALICE

[47] A. Fassò, A. Ferrari, P.R. Sala, G. Tsiledakis, Implementation of Xenon capture gammas in FLUKA for TRD background calculation, ALICE-INT-2001-28.

[48] http://root.cern.ch/root/doc/RootDoc.html

[49] P. Billoir; NIM **A225** (1984) 352, P. Billoir *et al.*; NIM **A241** (1985) 115, R.Fruhwirth; NIM **A262** (1987) 444, P.Billoir; **CPC** (1989) 390.

[50] CERN/LHCC 2005-049, ALICE Physics Performance Report, Volume 2 (5 December 2005);

[51] V. Karimäki, CMS Note 1997/051 (1997).

[52] http://alien.cern.ch/download/current/gClient/gShell_Documentation.html

[53] http://glite.web.cern.ch/glite

[54] http://root.cern.ch/root/PROOF.html

[55] CERN/LHCC 99-12.

[56] CERN/LHCC 2000-001.

[57] A. Dainese, PhD Thesis, University of Padova, 2003, [arXiv:nucl-ex/0311004].

[58] A. Stavinsky *et al*, NUKLEONIKA **49** (Supplement 2) (2004) 23–25; http://www.ichtj.waw.pl/ichtj/nukleon/back/full/vol49_2004/v49s2p023f.pdf

[59] P.K. Skowroński for ALICE Collaboration, [arXiv:physics/0306111].

[60] R. Lednický and V.L. Lyuboshitz, Sov. J. Nucl. Phys. **35** (1982) 770.

[61] `http://www.nscl.msu.edu/ pratt/freecodes/crab/home.html`

[62] C. Loizides, PhD Thesis, University of Frankfurt, 2005, [arXiv:nucl-ex/0501017].

[63] P.Skowronski, PhD Thesis.