EPJ Plus

EPJ.org
your physics journal

Eur. Phys. J. Plus (2011) **126**: 60 DOI: 10.1140/epjp/i2011-11060-6

The architecture of MEG simulation and analysis software

P.W. Cattaneo, R. Sawada, F. Cei, S. Yamada and M. Schneebeli

EDP SCIENCES Società Italiana di Fisica Springer

Eur. Phys. J. Plus (2011) **126**: 60
DOI 10.1140/epjp/i2011-11060-6

THE EUROPEAN PHYSICAL JOURNAL PLUS

Regular Article

The architecture of MEG simulation and analysis software

P.W. Cattaneo[1,a], R. Sawada[2], F. Cei[3], S. Yamada[4], and M. Schneebeli[5,6,b]

[1] INFN, Sezione di Pavia - Via Bassi 6, I-27100 Pavia, Italy
[2] ICEPP, University of Tokyo - 7-3-1 Hongo, Bunkyo-ku, 113-0033, Tokyo, Japan
[3] INFN and Department of Physics of the University of Pisa - Largo B. Pontecorvo 3, I-56127 Pisa, Italy
[4] KEK, High Energy Accelerator Research Organization - 1-1 Oho, Tsukuba, 305-0801, Ibaraki, Japan
[5] Paul Scherrer Institute PSI - CH-5232 Villigen, Switzerland
[6] Swiss Federal Institute of Technology ETH, CH-8093 Zürich, Switzerland

Received: 8 March 2011 / Revised: 17 June 2011
Published online: 5 July 2011 –

Abstract. MEG ($\mu^+ \to e^+\gamma$) is an experiment dedicated to the search for the $\mu^+ \to e^+\gamma$ decay that is strongly suppressed in the Standard Model, but allowed in many alternative models and therefore very sensitive to new physics. The offline software is based on two frameworks. The first is REM in FORTRAN 77, which is used for event generation and the detector simulation package GEM. The other is ROME in C++, used for the readout electronics simulation Bartender and for the reconstruction and analysis program Analyzer. Event display in the simulation is based on GEANT3 graphic libraries and in the reconstruction on ROOT graphic libraries. Data are stored in different formats at various stages of the processing. The frameworks include utilities for I/O, database access and format conversion transparent to the user.

1 Introduction

The MEG experiment at Paul Scherrer Institute (PSI) in Switzerland is searching for the rare decay $\mu^+ \to e^+\gamma$, employing a very intense ($3 \times 10^7 \mathrm{s}^{-1}$) μ^+ beam, which is stopped in a thin target at the center of the detector. MEG is a small-size collaboration ($\approx$ 50–60 physicists at any time) with a life span of about 10 years.

The collaboration started the software development in 2002 after a few years of prototype studies, with the goal of being ready for data taking in a technical run foreseen after 3 years. Since the beginning, the tight time schedule and the limited human resource available, in particular in the offline architecture group, emphasized the importance of reusing the software developed during the prototype studies and exploiting the existing expertise. Therefore great care has been devoted to provide a simple system that hides implementation details to the average programmer. That has allowed many members of the collaboration with limited programming skills to contribute to the development of the software of the experiment.

The detector consists of a Liquid Xenon Calorimeter for measuring the γ momentum vector and timing and of a spectrometer consisting of a set of drift chambers and of a timing counter embedded in a strong gradient magnetic field generated by a superconducting magnet (COBRA) for the measurement of e^+ kinematic variables. A sketch of the apparatus is shown in fig. 1. The waveforms from readout electronics are digitized at a $\approx$ 1 GHz frequency and stored in the output to optimize time resolution [1].

Waveform data is encoded in a format developed in the MEG group. The data of each channel consist of a header and binary waveforms. Each header contains a hardware channel number and parameters needed to decode data. The data can be encoded in different ways depending on the required compression factor, precision and characteristics of waveforms of each subdetector. The experiment totals $\approx$ 3000 channels and a reduction by a factor of 3 in data size is achieved applying zero suppression, waveform resampling or restricting the recorded region depending on the subdetector.

The typical DAQ event rate is $\approx$ 6 Hz. Data size is about 4.8 GB per run for 2000 events. Data files are compressed in the offline cluster by a factor of 2. Event size after the compression is 1.3 MB/event.

[a] e-mail: paolo.cattaneo@pv.infn.it
[b] *Present address:* DECTRIS Ltd. - Neuenhoferstrasse 107, CH-5400 Baden, Switzerland.

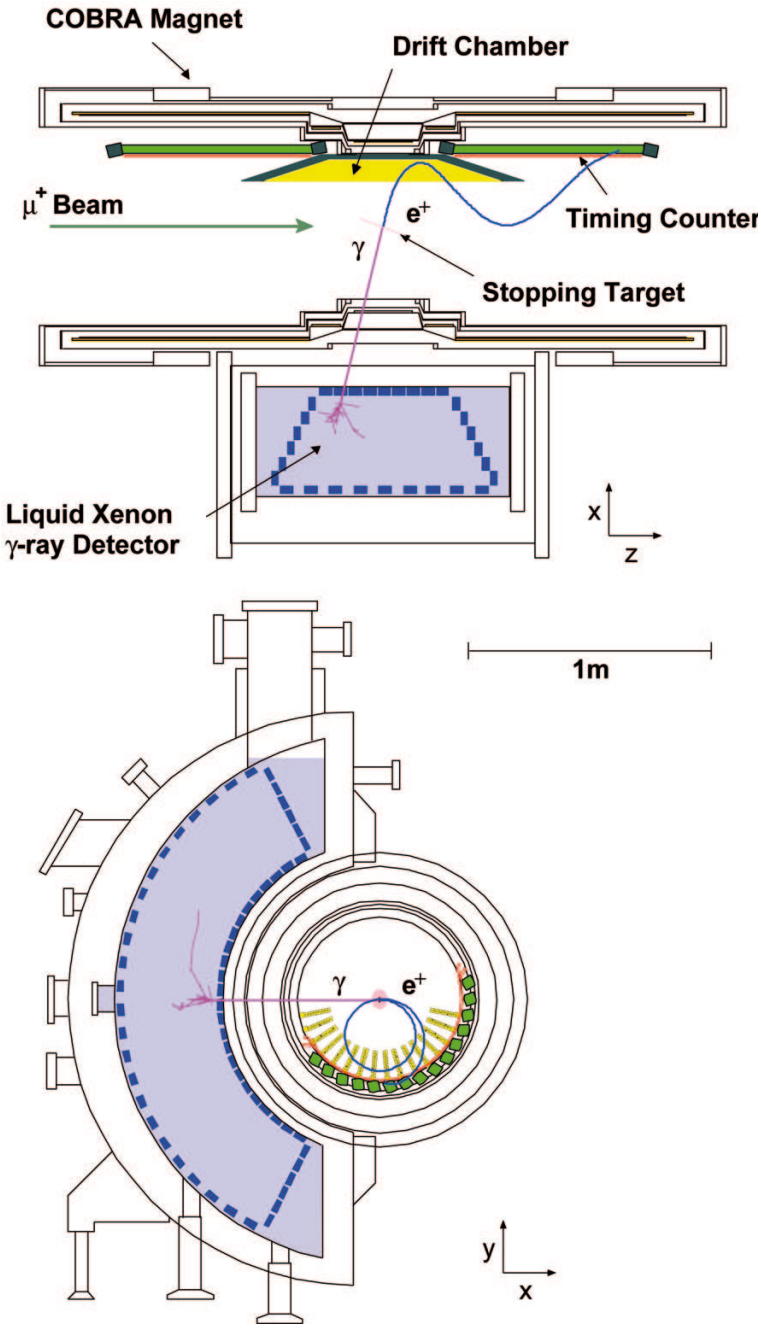

Fig. 1. The MEG experimental setup.

During ≈ 3 months in 2010, $\approx 21 \times 10^6$ $\mu^+ \to e^+\gamma$ triggers were collected for a total of 60 TB of data written on disk, half of which from physics runs and the rest from calibration runs.

The software requirements include the simulation of the generation of signal and background events, of their interaction with the detector and of the readout, the reconstruction from raw data, real or simulated, to high-level objects, *e.g.*, tracks and photons as well as providing an analysis environment.

The average time for simulating the interaction of a signal event in the detector is 5.8 s/event, while the average time for simulating the readout electronics is 1.2 s/event. The average time for the reconstruction is 1.6 s/event.

The software organization designed to comply with these requirements is presented.

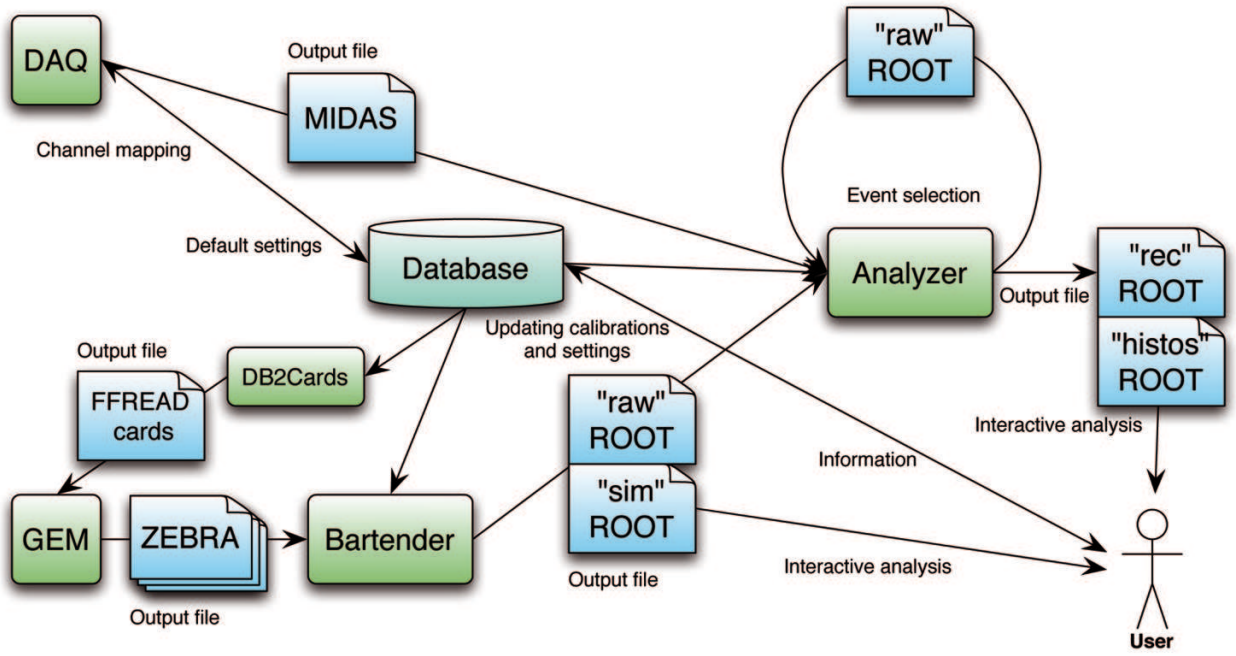

Fig. 2. Relations between MEG software components.

2 The MEG software structure

The MEG offline software consists mainly of GEM (event generation, particle tracking and detector simulation), Bartender (event mixing and electronics simulation) and Analyzer (reconstruction and analysis of experiment and simulation data). The relations between the various software components are shown in fig. 2.

The parameters in use in the programs are managed through a common SQL database.

The MEG DAQ system is based on MIDAS [2]; raw experimental data are therefore saved as binary files in the native format of the system. Briefly, the MIDAS format consists in an event header followed by MIDAS banks. Each bank is defined by a 4-character name and contains a description of the unique data type and an array of data.

The DAQ system inserts run information and default analysis parameters into the database when a run is taken.

These files are read by Analyzer that reconstructs the events and produces two files: a `rec` ROOT [3,4] file, which contains a ROOT `Tree` and a histogram file for quick data checks. Before Analyzer starts a run, analysis parameters are read from the database. The analysis parameters (geometry, calibration etc.) can be changed later by users and data can be reprocessed with the updated parameters. If necessary, Analyzer copies raw data of selected events (cut for physics analysis) into `raw` ROOT files for future reprocessing.

The simulation program GEM, steered by configuration files created by the DB2Cards program by reading the database, generates various types of events that are propagated in the detector. It is based on GEANT3 and CERNLIB and outputs data in exchange ZEBRA format [5]. Bartender reads those files and simulates the readout electronics to convert hits into waveforms. The simulated waveforms are written in `raw` ROOT files whose bank structure is the same as experimental data in MIDAS files. In `sim` files simulation, specific variables, such as kinematics of generated particles and true hit information, are saved. Analyzer reconstructs events from `raw` files using the same algorithms as for the experimental data. High-level physics analysis is also realized within Analyzer.

Version control is managed by the Subversion [6] package.

3 REM: a FORTRAN 77 framework

As anticipated above, the technical choices in designing the offline architecture were driven by considerations about the time schedule, the manpower and the technical skills available in the collaboration at the start of the project. The existence of important fragments of simulation code developed in FORTRAN 77 and GEANT3 during the prototype phase at the time of the choice motivated the collaboration to retain the programming language and the library for the simulation of the experiment.

Nevertheless, the simulation software was organized following a modern programming paradigm, that is using an Object-Oriented (OO) approach organized in a framework [7].

3.1 Implementation of a FORTRAN 77 framework

The detector simulation section GEM of the MEG software is written in FORTRAN 77, that was designed for procedure-oriented structured programming, not for OO programming.

Nevertheless, a programming paradigm can be implemented in a variety of programming languages, even not designed for it. A limited but satisfactory support to the OO paradigm is within reach also in FORTRAN 77 on the basis of the following list of approximate equivalences between procedure-oriented and OO concepts:

- Class ↔ Library;
- Class data ↔ Data structure (FORTRAN 77 Common block);
- Class interface ↔ Set of library routines;
- Base Class ↔ Module standardization;
- Virtual Class ↔ Alternate choice of libraries.

3.2 Modules

The Module is the basic unit manipulated by the framework that corresponds to an OO class. Each Module is implemented concretely in a library. There are different types of Modules, that can be classified as folows:

- Basic Module: empty Module.
- Steerable Module: Module steerable by configuration files (cards).
- Data Module: contains only data.
- Algorithm Module: implements an algorithm using other Modules.
- Service Module: provides interface to external libraries.

These types share a common set of routines and differ by additional functionalities depending on the Module type implementing the OO paradigm of class hierarchy.

3.3 The framework: REM

The framework is a Module with an event loop. The Modules associated to the framework are accessed in sequence by calling their routines in the corresponding framework routines.

Three module are provided by default in REM:

- Steering cards: FFREAD package.
- I/O: ZEBRA I/O.
- Histogramming: HBOOK package.

The other Modules are project-dependent and their routines are called in the corresponding framework user routines. These user routines, provided empty by default, are called by the framework routines. They can be overwritten implementing the OO inheritance mechanism.

4 GEM: the Monte Carlo simulation

The propagation of the μ^+ beam in the last section of the beam line, its interaction in the target, the particle decay and the propagation and interactions of the decay products in the detector are simulated with a FORTRAN 77 Monte Carlo program (GEM) based on the GEANT3 package [8]. GEM can generate several event types, such as $\mu^+ \to e^+\gamma$ signal (shown in fig. 3), radiative muon decay, Michel muon decay, cosmic ray, alpha source calibration and many others. GEM incorporates a detailed description of the material and simulates the interactions of the particles in the detector as well as the response of the sub-detectors up to the readout stage. In particular, the photon propagation in the Liquid Xenon Calorimeter and in the Timing Counter is simulated in detail.

The program is heavily modularized using the FORTRAN 77 framework REM. This approach simplifies the addition of new Modules; Modules can be either sub-detector simulation sections or service tools like, *e.g.*, graphics.

Within this approach, the GEANT3 library can be treated as a Module and sequenced like any other module.

GEM is steered by configuration files, called cards, read by the FFREAD package [9], that is available in REM. These cards can be generated through the DB2Cards that is a ROME based framework. DB2Cards reads parameters from the database and output FFREAD cards, one for each Module, under the control of a XML configuration file. This file permits to select the simulation configuration, *e.g.*, year dependent or calibration setups, that are maintained in the database.

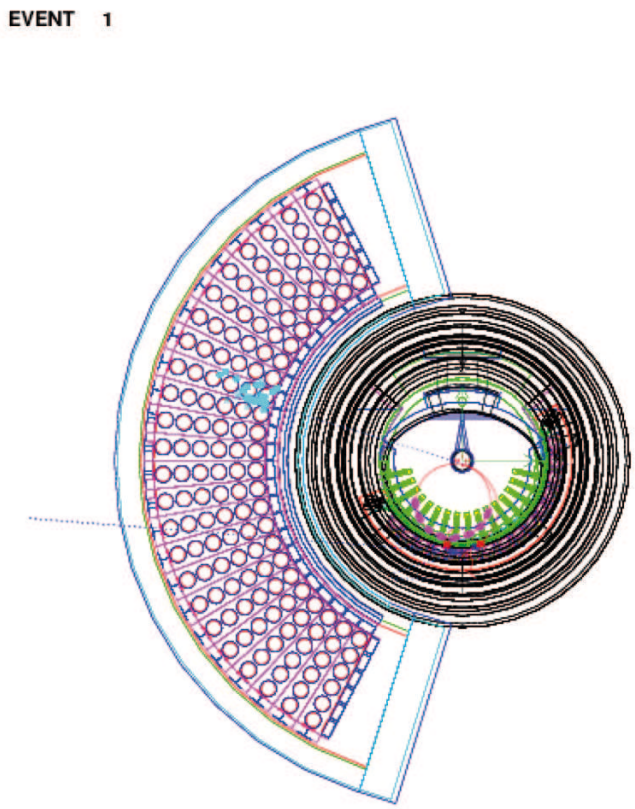

Fig. 3. A $\mu^+ \to e^+\gamma$ simulated event: the e^+ track is in red with hits in drift chambers and timing counter in violet, red and blue, the γ track in blue with hits in the Liquid Xenon Calorimeter in cyan.

The most natural choice for the format of the GEM output files is ZEBRA. Potential disadvantages of this approach are that the manipulation of ZEBRA banks is not user friendly, error prone and requires a significant knowledge of the package. A solution to these problems consists in manipulating only variables in common blocks in the code and then mapping these variables into the output ZEBRA banks. This is done automatically by providing a bank description based on the DZDOC format [5] and generating the following routines for each bank `xxxx` through a Perl [10] script:

`get_xxxx`	fetch the bank link,
`print_xxxx`	print out of the bank,
`build_xxxx`	fill the bank with the variables in common block (before writing out),
`fill_xxxx`	fill the common block variables with bank content (after reading in).

GEM provides for each Module `yyyy` the routines `fill(build)yyyyrunheader` and `fill(build)yyyyeve` that call all the corresponding routines of the banks related to the module. GEM provides also the routines `fill(build)gemrunheader` and `fill(build)gemeve` that call the corresponding routines for all the Modules.

The `buildgemrunheader` is called once per run and `buildgemeve` is called once per event to build the banks from the variables in common blocks before calling the I/O ZEBRA routines in REM.

5 The database

Run-dependent information such as geometry, calibrations and analysis parameters are stored in a relational database, used for the DAQ frontend, analysis and simulation. Online data logger inserts immediately an entry into the database when a run is taken. A run can be processed by Analyzer with the default settings and reprocessed later with improved calibration constants after modifying the database. For simulation, the dedicated program DB2Cards reads the database and write the FFREAD cards required by GEM for all the configurations required. Therefore, all packages use consistently a common database.

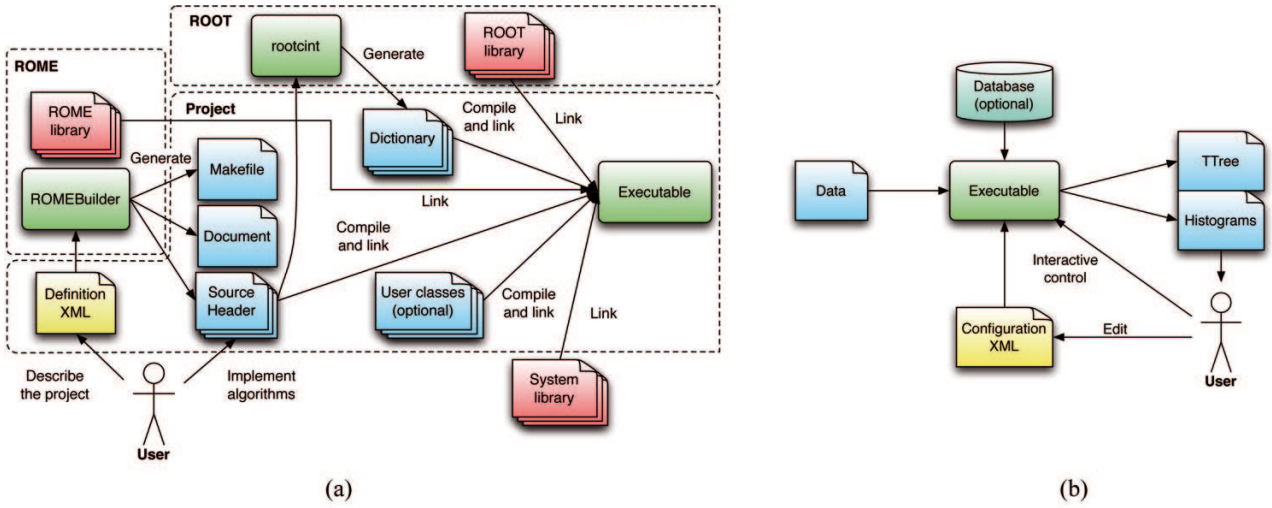

Fig. 4. Components in the ROME environment (a) at build time, and (b) at run time.

For the main database, MySQL [11] is used so that clients can connect over the network. Daily snapshots are taken in MySQL script format and SQLite [12] format. SQLite is a single-file database; therefore it can be used without network, and can be used for test purposes by modifying local copies without affecting other users. Information on all the runs and all the simulation configurations are stored in the database. The MEG database consists of a few hundreds tables and each has a direct or indirect relation to the mother table `RunCatalog` so that a run number suffices to retrieve all the information, and no recompilation or manual modification of configuration files is required to analyze any run sample. At the end of May 2011, the size of the MySQL database was ≈ 750 MB.

6 ROME: A framework generator

ROME [13,14] is a ROOT-based framework generator for event-based data processing. It has been developed in the MEG collaboration but has been designed as a general-purpose software so that it can be used for other experiments too.

The key concept of ROME is to generate most of the code of a project, except the analysis (or simulation) algorithms.

In general, data processing software consists of three parts: the first is a project-independent part such as, *e.g.*, user interface, handling of the event loop. The second is a project-dependent part, which can be summarized in a compact way such as, *e.g.*, data structure and calling sequence of algorithms. The third is a completely project-dependent part such as, *e.g.*, the implementation of analysis algorithms.

Figure 4 shows the components in the ROME environment. In this environment, the first part is included in the ROME package, and also the ROOT infrastructure is used. For the second part, a programmer describes the framework for his/her experiment in a clear and compact way in a XML definition file. Out of this file, `ROMEBuilder` program generates all experiment specific classes and modifies the framework. It calls also the `make` command after the source code generation; therefore the build procedure shown in fig. 4(a) can be done with a single command. For the third part, a programmer adds the algorithm code to the pre-generated methods. Further modifications can be done by editing the definition XML file or by modifying algorithm implementations, then running `ROMEBuilder` again.

Because of the generation scheme, the amount of hand-written code becomes smaller, and it becomes possible to start or modify the software without learning the complicated implementations of the framework.

The generated framework is linked with the ROOT libraries; therefore all ROOT classes are available for the analysis. Additional classes written by hand can be also linked. The generated program is steered using a configuration XML file at run time. An interactive control of the program, for example pausing the event loop and plotting histograms, is possible.

The following list is part of the items automatically generated by ROME according to a XML definition file:

- Data classes (`Folders`) with a complete set of methods.
- Algorithm classes (`Tasks`) with empty methods to be filled by a programmer.
- Visualization classes (`Tabs`) with empty methods to be filled by a programmer.

```
<Folder>
   <FolderName>Photon</FolderName>
      <ArraySize>10</ArraySize>
      <Field>
         <FieldName>Energy</FieldName>
         <FieldType>Double_t</FieldType>
         <FieldComment>Energy of a photon</FieldComment>
      </Field>
      <Field>
         <FieldName>Time</FieldName>
         <FieldType>Double_t</FieldType>
         <FieldComment>Time of a photon</FieldComment>
      </Field>
</Folder>
```

```
class MEGPhoton : public TObject
{
protected:
   Double_t  Energy;  // Energy of a photon
   Double_t  Time;    // Time of a photon
...

public:
   MEGPhoton(Double_t EnergyV=0, Double_t TimeV=0);
   virtual ~MEGPhoton();
...
   Double_t GetEnergy() const
               { return Energy; }
   Double_t GetTime() const
               { return Time;   }
   void  SetEnergy(Double_t Energy_v)
               { Energy = Energy_v; }
   void  SetTime(Double_t Time_v)
               { Time   = Time_v;   }
...
}
```

Fig. 5. An example of `Folder` definition in a XML file (upper) and part of the C++ code generated by ROME (lower).

- Data input classes to read user defined data files with empty methods to be filled by a programmer.
- Code to create and write histograms. The histograms can be filled in user code.
- Code for I/O of `TTrees`[1] into files.
- Code to read and write configuration XML files.
- Code to read and write SQL database. MySQL, PostgreSQL [15] and SQLite are supported and switchable by a configuration file at run time.
- Code to read MIDAS format files and to connect to MIDAS Online Database System (ODB) to access online data.
- Makefile is automatically generated or updated when new classes are defined by a definition XML.
- HTML document where the description of `Task`s and that of each variable in `Folder`s are written. ROOT style document, like "reference guide" in the ROOT web page can be also generated for user code.

ROME implements the organization commonly used in OO applications in high-energy physics [16]: *data objects*, whose function is to store data, are separated from *algorithm objects*, whose function is to incorporate algorithms.

The former are implemented as a `Folder` class, the latter as a `Task` class. `Task`s are derived from ROOT `TTask`; therefore recursive calling sequence is realized. ROME `Folder`s are derived from ROOT `TObject` (not from `TFolder`), and they can be filled into ROOT `TTree` as a single object or as an array in ROOT `TClonesArray`.

For `Folder`s, ROME generates not only the class itself, but also modifies the part of the framework related to the `Folder` such as allocation and initialization, adding or setting the address of a branch in a `TTree` for writing (reading) the `Folder` to (from) a file, filling the variables by reading the database at the beginning of a run (if required in XML).

[1] TTree is the ROOT implementation of the data structure tree concept.

```
<Task>
   <TaskName>PhotonAnalysis</TaskName>
   <SteeringParameters>
      <SteeringParameterField>
         <SPFieldName>DebugPrint</SPFieldName>
         <SPFieldType>Bool_t</SPFieldType>
      </SteeringParameterField>
   </SteeringParameters>
</Task>
```

```
...
void MEGTPhotonAnalysis::Init()
{
}

void MEGTPhotonAnalysis::BeginOfRun()
{
}

void MEGTPhotonAnalysis::Event()
{
...
  if (GetSP()->GetDebugPrint()) {
    for (int i=0;i<10;i++) {
      cout
        <<gAnalyzer->GetPhotonAt(i)->GetEnergy()
        <<endl;
    }
  }
...
}

void MEGTPhotonAnalysis::EndOfRun()
{
}

void MEGTPhotonAnalysis::Terminate()
{
}
...
```

Fig. 6. An example of `Task` definition in a XML file (upper), and part of the C++ code generated by ROME (lower).

```
<Tree>
   <TreeName>DataTree</TreeName>
   <Branch>
      <BranchName>PhotonBranch</BranchName>
      <RelatedFolder>Photon</RelatedFolder>
   </Branch>
</Tree>
```

Fig. 7. An example of `Tree` definition in a XML file.

A definition of a `Folder` reads like the XML document shown in fig. 5 together with part of the C++ code generated by ROME, according to the definition. This `Photon` instance has two variables, `Energy` and `Time`. The generated class has these variables as its data members, and `Set` and `Get` methods are defined. The framework generates automatically, for example, 10 instances (the number can be fixed or variable) at the beginning of the program and those instances are available in the user code out-of-package. For example, `GetPhotonAt()` and `GetEnergy()` shown in fig. 6 are generated according to the description in the XML definition file of the `Folder` without manual programming. Any

types of `Field`, both fundamental and derived, can be added in the `Folder` structure as far as it is supported by ROOT dictionary generation (dictionaries are needed for `TTree` I/O or socket connection over the network).

A definition of an algorithm object, that is a `Task`, reads like the XML document shown in fig. 6. According to the definition file, ROME generates header and source files. A generated source file has empty methods, and a programmer can implement the analysis in it immediately. As an example, in the code in fig. 6, a few lines to access a `Folder` are added to the generated file. ROME generates not only the task class itself, but modifies the framework to call it in an order specified in the definition XML. In this example, a configuration parameter `DebugPrint` can be changed using a configuration XML file at run time without re-compile. A function call `GetSP()->GetDebugPrint()` shown in the example code is available without any manual programming, and a field to configure the parameter automatically appears in a configuration XML file after the first use of the file.

The framework outputs one or more `TTree`s. A programmer can define `Tree`s and add `Folder`s to it as branches in a XML description file. The framework code is automatically modified; therefore no manual programming is needed to add branches to be read or written. Figure 7 shows an example of `Tree` structure.

Output files can be used for an interactive analysis, and further analyzed by ROOT macros.

Output files of each step can be used as input files of the following step; therefore the analysis can be separated into several steps. For example, in the analysis of MEG, we can save the results of the waveform analysis, which is the most time-consuming in the chain, and perform reconstructions on this file to improve the algorithm many times without redoing the waveform analysis.

An interactive mode, which is almost the same as ROOT interactive mode, is also provided. In the interactive mode or in macros, experiment specific classes are also available in addition to the standard ROOT classes.

ROME also generates a HTML document and a Makefile. The generated framework is already compilable just by the `make` command and, after that, is executable.

The generation mechanism is used not only at the beginning of the project, but also during the code development. For example, a programmer can easily add a new configuration parameter to an existing `Task`, or add new variables to a `Folder`. The code in the framework is automatically modified consistently.

MEG Analyzer consists of about 200 `Folder` classes and 100 `Task` classes. The total number of lines in the Analyzer code is more than one million. 84% of them are either generated by `rootcint` [3] or ROME, or included in the ROME package, while the rest was written manually.

7 Readout simulation and event mixing

Following the detector simulation and before the reconstruction and analysis program, an intermediate program, called Bartender, is required for the processing of Monte Carlo data. This program serves different roles:

- Conversion of ZEBRA files into ROOT files.
- Readout simulation.
- Event mixing.

It reads the GEM output ZEBRA files calling `fillgemrunheader` once per run and `fillgemeve` once per event after calling the I/O ZEBRA routines to fill the variables in FORTRAN common blocks from the banks. These variables are finally mapped to C++ classes manually.

Simulation specific data such as kinematics of generated particles, true hit information, etc., can be streamed in a `sim Tree` in separate ROOT files for further studies.

It simulates detector readout electronics and produces waveforms. For example, the Liquid Xenon Calorimeter waveforms are obtained by the convolution of the single-photoelectron response of a photomultiplier tube (PMT) with hit-time information of scintillation photons simulated in GEM. PMT amplification, signal attenuation, saturation of the readout electronics, noise, etc., are taken into account. Simulated waveforms are encoded in the same manner as the experimental ones and written in a `raw Tree` in ROOT files.

It makes a mixture of several sub-events; rates of each event type are set with a configuration file. To study the combinatorial background events, sub-events are mixed with various relative timing with respect to each other and with respect to the trigger. For instance, random and fixed timing can be selected, and this allows simulating many different pile-up configurations with a limited number of samples of events simulated through the detector.

8 The reconstruction and analysis program

Analyzer incorporates multiple purposes: event reconstruction, visualization, computation of calibration constants and physics analysis.

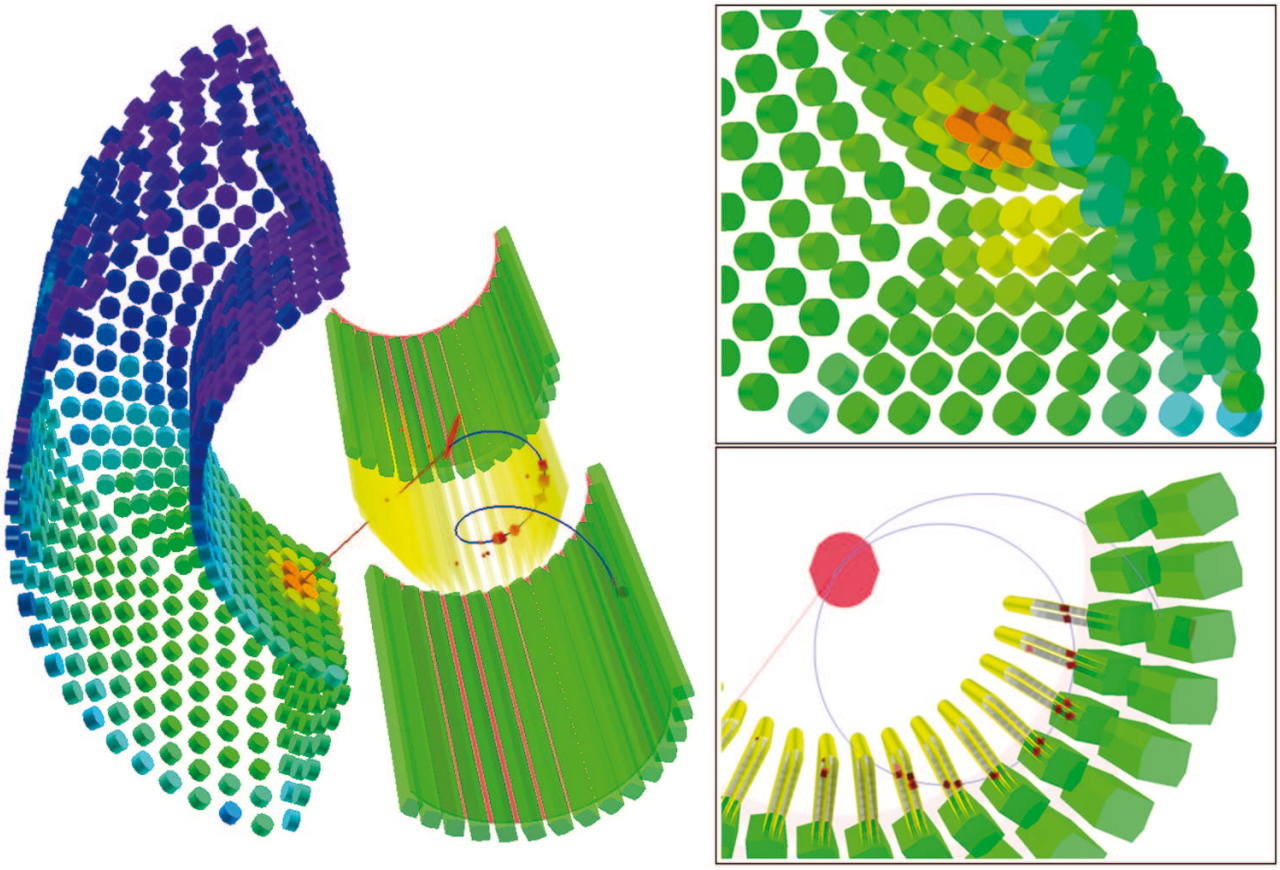

Fig. 8. A $\mu^+ \to e^+\gamma$ reconstructed event and closer views. Reconstructed hits in drift chambers and timing counters, a positron track and a γ-ray are shown. The color code of the Calorimeter PMTs represents the output of each PMT.

8.1 Event reconstruction

Analyzer consists of several `Task`s for each step of the analysis; each `Task` can be switched on/off.

In the first step, raw data are read and calibrations are applied to waveforms. In the second step, waveform analysis specialized for each sub-detector are performed to extract time and charge of pulses. Waveforms are also used to identify pile-up events and for particle identifications.

In the third and last step, the events are reconstructed using algorithms implemented by experts of each sub-detector. Several different algorithms are implemented to reconstruct each kinematic parameter for crosschecks. Each `Task` may have a dedicated `Folder` to write its result. `Task`s share a `Folder` to hold results of a standard choice among those algorithms; this choice is specified by a configuration file. `Task`s are executed in the same process and results are written in an output file together.

Figure 8 shows a reconstructed experimental event.

8.2 Visualization

Data quality is monitored for various time spans: event-by-event, run-by-run or in days.

For event-by-event monitoring, several displays are implemented. Figure 9 shows one of them. The displays show waveforms, status of trigger, reconstructed hits and tracks and any other information useful for monitoring. Those displays are used for both online and offline. When it is used for online monitoring, Analyzer and DAQ run in parallel and data are transferred over a socket connection. Hard copies of the displays are saved periodically for remotely monitoring using web browsers.

Two types of portable document format (PDF) files are automatically prepared by macros, which read histogram files made by Analyzer. The first type shows histograms to describe the run and is made automatically for each run soon after the run is finished. The second type shows strip charts to monitor time variations of the status of the detector and of the electronics in a day or a week.

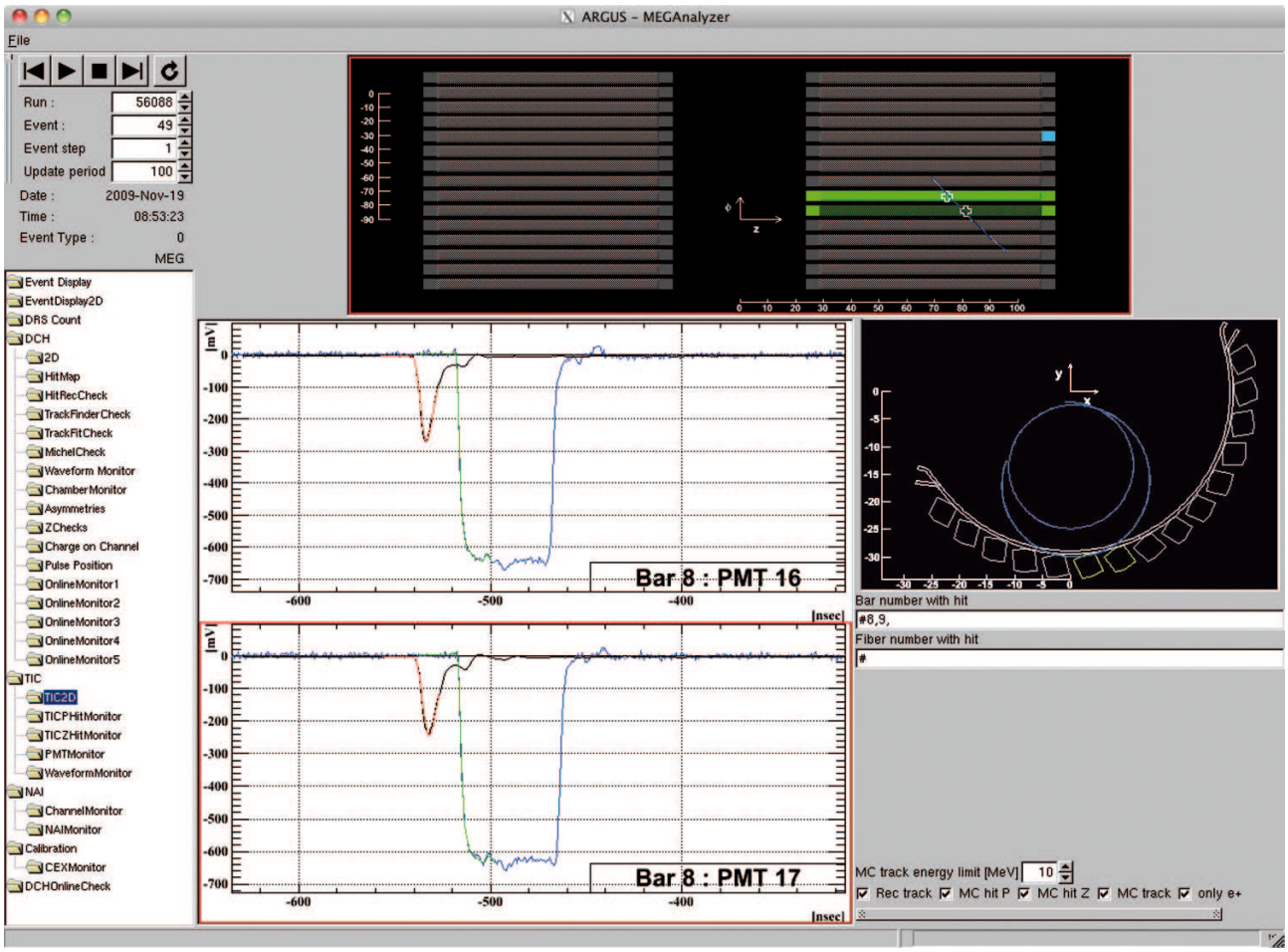

Fig. 9. A graphical display of timing counter hits, waveforms. A reconstructed positron track is also shown.

8.3 Calibration

Analyzer is used also to compute calibration constants (photomultiplier gains, time offsets, etc.). Each calibration constant is associated to a `Task`. The calibration `Task`s are usually run on events already processed with a preliminary set of calibration constants. The updated calibration constants can be made available in a variety of format: histograms, text file or SQL macro. They can be stored in the database, and used in the next round of reconstruction.

8.4 Physics analysis

Event preselection and blinding for physics analysis, described in sect. 9, are implemented in Analyzer. On events in the analysis region, likelihood analysis is performed to calculate the best estimate of the number of $\mu^+ \to e^+\gamma$ signal candidates, its confidence interval and the significance. The 90% confidence interval of the number of signal events is calculated using the unified approach [17]. We made independent likelihood analysis tools with different statistical methods or parametrization of probability density functions for crosschecks.

9 Offline processing

Just after a run is taken, a raw data file written in the MIDAS format is sent to the offline cluster, and a process to analyze it starts automatically. The MIDAS files are compressed and stored on tapes. The compressed MIDAS files and `rec` files of calibration runs are accessible for further studies, while a special treatment is done for the data of physics runs.

MEG has adopted the principle of "blind" analysis in searching for the $\mu^+ \to e^+\gamma$ signal. This means that the events with kinematic parameters closest to the expected signal (in the "blinding box") cannot be used for determining the analysis parameters (*e.g.*, the cuts or the probability density functions) to avoid biasing the analysis. In order to guarantee that, the data in the "blinding box" are inaccessible during the first phase of the analysis.

This concept is realized in Analyzer with `Task`s streaming the events into different ROOT files depending on the selection criteria they satisfy.

A first round of processing operates a pre-selection on coarsely calibrated data with loose cuts that are streamed in:

`selected` : events passing the pre-selection,
`unselected` : events not passing the pre-selection,
`unbiased` : all calibration trigger events and every fiftieth physics-trigger event.

`Trees` containing raw waveforms are produced for "selected" and "unbiased" events in this step. The "unbiased" samples are used for monitoring of the experiment and for the calibrations. The "selected" events are not accessible. After the calibrations are finalized, reconstruction is performed on the "selected" samples using `raw` files. At the end of this step, another `Task` applies tighter cuts defining the "blinding box". The events are streamed into the files:

`blind` : events preselected in the "blinding box", candidate to be signal,
`open` : events preselected but outside the "blinding box",

and the "selected" files are deleted. The "blind" files are made accessible only when the analysis is finalized.

10 Conclusion

Software is a crucial component of any experiment and its power and flexibility is a key ingredient of its success.

MEG had the challenge to design a software structure that could strike a balance between flexibility and user-friendliness. The limited size of the offline architecture group and the requirement that a large fraction of the collaboration could contribute to the programming of the algorithms, have led to greatly emphasize the use of known packages as well as the shielding from the average programmer of I/O handling, format conversion and Object-Oriented programming into the frameworks.

A mixed language environment with two separate frameworks, one for each environment, proved to be successful. It relies heavily on standard software elements like GEANT3, ZEBRA, FFREAD in the simulation section implemented in FORTRAN 77; XML, ROOT in the rest of the code implemented in C++; MySQL and SQLite for the database.

This configuration allowed the implementation of all experimental requirements within the tight time and manpower constraints, such to support the physics analysis first published in [18].

We acknowledge the role of Dr. S. Ritt from PSI, who is the main author of the online software MIDAS. Integration of each sub-detector part was done by many collaborators; forty of them have contributed to the MEG software.

References

1. S. Ritt, R. Dinapoli, U. Hartmann, Nucl. Instrum. Methods A **623**, 486 (2010); the 1st International Conference on Technology and Instrumentation in Particle Physics.
2. MIDAS: `http://midas.psi.ch`.
3. ROOT: `http://root.cern.ch`.
4. R. Brun, F. Rademakers, Nucl. Instrum. Methods. A **389**, 81 (1997).
5. R. Brun, J. Zoll, *ZEBRA - Data Structure Management System*, CERN Program Library Writeups Q100, 1995.
6. Subversion: `http://subversion.apache.org`.
7. R. Brun, Eur. Phys. J. Plus **126**, 14 (2011).
8. R. Brun *et al.*, *GEANT3 - Detector Description and Simulation Tool*, CERN Program Library Long Writeups W5013, 1993.
9. FFREAD, CERN Program Library Long Writeups Q123, 1993.
10. L. Wall, T. Christiansen, J. Orwant, *Programming Perl* (O'Reilly Media, Sebastopol, CA, 2010).
11. MySQL: `http://www.mysql.com`.
12. SQLite: `http://www.sqlite.org`.
13. ROME: `http://midas.psi.ch/rome`.
14. M. Schneebeli, R. Sawada, S. Ritt, "ROME - A universally applicable analysis framework generator", in *Proceedings of the International Conference on Computing in High Energy and Nuclear Physics (CHEP06), Mumbai, India, 2006.*
15. PostgreSQL: `http://www.posrgresql.org`.
16. G. Corti *et al.*, IEEE Trans. Nucl. Sci. **53**, 1323 (2006).
17. G.J. Feldman, R.D. Cousins, Phys. Rev. D **57**, 3873 (1998).
18. J. Adam *et al.*, Nucl. Phys. B **834**, 1 (2010).