Design of a F77 based framework: REM

Paolo Walter Cattaneo

INFN Pavia, Via Bassi 6 - 27100, Pavia, Italy
Email: Paolo.Cattaneo@pv.infn.it

Abstract

This note describe the design and implementation of the REM framework. It is based on concepts borrowed by OO programming paradigm even if it is implemented in F77, that is not a language designed for OO.
That requires some ingenuity and encounters some limitations but nevertheless the results are quite satisfying.
The principle underlying the design are detailed and the implementation techniques used to mimic OO are explained.
A section providing a step by step introduction for an end user is also provided.

1 OO in F77

Recent trends in programming favours the introduction of the Object Oriented (OO) paradigm [1].
Some of the main design criteria inspired by the Object Oriented (OO) paradigm are

- Modularity
- Module standardization
- Clear interface
- Independence on specific choices of external libraries

A host of new languages (or extension of old ones) have been developed to provide complete or partial support to the OO paradigm. These languages like C++, Java, Eiffel are often referred as OO languages.
At some extent, a limited support to the OO paradigm can be achieved also in languages designed for procedure oriented structured programming.
An approximate list of equivalences can be the following

- Class $\leftrightarrow$ Library
- Class data $\leftrightarrow$ Data structure (F77 Common block, F90 module, C structure, etc..)
- Class interface $\leftrightarrow$ restricted set of library routines
- Base Class $\leftrightarrow$ Module standardization
- Virtual Class $\leftrightarrow$ Alternate choice of libraries

Additional OO features can be supported with the help of scripts generating code for compiler processing.

1.1 Modules

We define the Module as the basic unit manipulated by the framework. Each module is implemented concretely in a library.
There are different types of Module. They can be classified as

- Basic Module : empty module
- Steerable Module : module steerable by cards
- Data Module : contains only data
- Algorithm Module : implement an algorithm using other modules
- Service Module : provide interface to external libraries

A Basic module, that is in some sense a base class in OO jargon, contains

- xxxini Initialize Module with default parameters
- xxxset Check cards and define extra parameters for Module
- xxxend End processing Module (close files, print statistics)

A Steerable module requires one more routines for getting input parameters from the user

- xxxcar Define cards related to Module

The data are supplied through the FFREAD package and this routines defines the cards with the parameters.
A Data module contains data to be manipulated. That requires furthermore a routine for defining histograms

- xxxhis Book histograms related to Module

Algorithm Modules incorporate an algorithm called each event, to transform the data in some other Data Modules.
They are Data modules themselves containing some data to steer the algorithm, but mainly consist of functions. An algorithm is defined by its input and output data, e.g. from raw data to hits, or from hits to tracks. The details of the algorithm itself must be hidden from the users within the module. In this way different implementations of the same algorithm can be used through recompilation.
The routine defining the algorithm is

- xxxrun Run the algorithm related to Module

The data are stored in common blocks, that are handled by include files. It is important to keep separated data to be manipulated only internally to the module (private data), that can be changed more easily, and data visible from external modules (public data).
Various modules derived from the previous modules can be built, e.g. the 'Geant' Modules used in GEANT, derived by Data Module contains also

- xxxdraw Draw the geometry of Module
- xxxmate Define materials for Module
- xxxinitmed Initialize Tracking media for Module

- xxxtmed Define media for Module
- xxxgeom Define the geometry of Module

A detector Module is derived by a 'Geant' Modules, containing also

- xxxdefhit Define the hit for Module
- xxxhit Calculate the hit for module
- xxxdigi Calculate the digitization for Module
- xxxout Assemble the output for Module

These Modules may also contain data under form of common blocks and parameters located in include files. These data should be internal and therefore not accessed from outside the Module itself. That allows a greater freedom to organize the include files.
Nonetheless for the GEANT Modules there is a standard given by

- xxxcflags.inc Module flags
- xxxcmate.inc Module materials
- xxxctmed.inc Module tracking media
- xxxcgeom.inc Module geometry

The need for some coherence between the data files comes from the requirements of defining material and tracking media with unique numbers throughout the whole program.
For detector Module there is also a

- xxxchit.inc Module hit

containing Module hit information.
Service Modules interface external libraries, e.g. histogramming. From the outside they should appear with a standard, library independent interface.
They are organized as a Steerable Module plus all the routines providing the interface to a given functionality (e.g. histogramming, IO). A specific implementation of this module will provide the library dependent calls.
A mechanism must be provided to switch between different libraries having the same interface. It is provided at the make level, through recompilation of the executable. It cannot be done at run time, as in true OO languages.

2 The framework *rem*

A framework (or sequencer or manager) is a program designed to access Module in sequence upgrading some Data Module through the algorithm contained in some Algorithm Modules making use of external library accessed through Service Modules.
The standard routines in the *rem* library, that correspond to a base class in OO jargon, are

- xxxini Initialize Sequencer with default parameters
- xxxcar Define cards of Sequencer

- xxxset Check cards and define extra parameters for Sequencer
- xxxhis Book histograms related to Sequencer
- xxxloop Run through events
- getfile Open input file
- xxxrunfirst Perform operation at run start
- xxxrun Apply Sequencer Algorithm to each event
- xxxwrite Write event after processing
- xxxrunlast Perform operation at run end
- dropfile Close input file
- xxxend End processing Sequencer (close files, print statistics)

The framework is an algorithm Module with the addition of the *xxxloop* routine where *xxx* is replaced by *rem*.
That is because the main task of the framework is to process in series some Modules. The Module routines are called in the corresponding framework routines.
Each of these routines except *xxxwrite* and *xxxloop* calls a routine with a *u* (User) after the prefix *xxx*.
Default empty routines are supplied in the library that can be overwritten by the user with application specific code. They are equivalent to a class derived through inheritance mechanism.
In order to establish the right linking order it is necessary that the user routines in the application are linked as object files and not put into a library.
If, as usual, the application code is split between a directory with the library code and a directory with code incorporated only in the executable, the framework user routines go in the latter.

2.1 The *rem* event structure

The data handled by *rem* are organized in runs and events.
If the *rem* application processes multiple input file, each run corresponds to a input file.
Each run consists in a run header followed by a set of events.
Both the run header and the event structure consists of a base including some basic data and links made available by *rem* itself and an application part with user supplied data and links.

3 The main framework routine

The file containing the main PROGRAM is compiled outside the library and must be linked separately as an object file. That because the F77 linking rule does not allow the main to be in a library.
This routine calls, in sequence

- remini Initialize Sequencer with default parameters
- remcar Define cards of Sequencer

- remset Check cards and define extra parameters for Sequencer
- remhis Book histograms related to Sequencer
- remloop Loop through files and events applying Sequencer Algorithm
- remend End processing Sequencer (close files, print statistics)

The remloop routine includes two nested loops. The outer one on the input files and the inner one on the events. If input files exist, each corresponding to a run, REM reads events from them otherwise generates events internally. After that an event is available, it is processed by remrun and written to disk.

4 Default and user modules

Some default Modules are called in the base Modules routines: card, histogramming and IO.

- Card: this Basic Module is based on FFREAD [2]. The user must supply for each user module the xxxcar.F with the calls to FFKEY for defining the input cards and mapping them to commons in include files. Usually one card corresponds to a common that corresponds to an include file.
- Histogram: this Steerable Module is based on HBOOK [3]. It is steered by the FILEHIS card. IF present, HBOOK is initialized and the corresponding file is opened before the loops and is closed at the end. Histogram booking should take place in the Module xxxhis routine or in the remurunfirst routine. Filling is done anywhere in the user Module routines.
- IO: the IO default package is based on ZEBRA [4]. It is steered by the FILEIN and FILEOUT cards for input and output files respectively, the details in the Appendix.
  Each REM based application *zzzz* with IO must provide four files in the corresponding libraries
  - fillzzzzeve.F Fill event common blocks from ZEBRA banks
  - fillzzzzrunheader.F Fill run header common blocks from ZEBRA banks
  - buildzzzzeve.F Build event ZEBRA banks from common blocks
  - buildzzzzrunheader.F Build run header ZEBRA banks from common blocks

  The *fill* routines are used in input for reading events written by *zzzz*, the *build* are used in output for writing events processed by *zzzz*. Therefore *buildzzzzeve* must be called by builduevent (in the executable directory) and *buildzzzzrunheader* by buildurunheader.
  If the application is reading data from the application *wwww*, the routine *fillwwwweve* must be called by filluevent (in the executable directory) and *fillwwwweve* must be called by fillurunheader.
  More details on the IO mechanism are available in [5].

Other Modules have to be called in the corresponding user routines.
Furthermore, within the framework there are two nested file and event loops (remloop). Within the outer loop, each input file is opened, then the routine *remrunfirst* is called, then in the inner loop an event is fetched to be processed by the routine *remrun*, followed by the routine writing the event to disk *remwrite*.

After all events in a file are processed the routine *remrunlast* is called and then the routine *dropfile* close the file.

5 Applications

5.1 IO tool

The most straightforward application is using REM simply for IO starting from a file generated by another REM application *xxx*. It is assumed that this application produces a library *libxxx* containing the routines fillxxxeve and fillxxxrunheader.
The first step is creating a directory *aaa* with sub directory

- *cards* Containing the steering cards
- *aaa* Containing the build and fill user files

A standard Rules Make file must be put in each sub directory and in the sub directory *aaa* Make file the library libxxx.a must be added to the link list of *locallibs*, the executable target PROD=*aaa* must be defined and the main rem object must be linked adding it to DIR_OBJS.
Then the files filluevent.F and fillurunheader.F must be copied in the directory *aaaa* from the library iolib. These files must call respectively the routines fillxxxeve and fillxxxrunheader.
A card file *io.cards* in the *cards* directory must contain the input and output files. If an output file is required, the files builduevent.F and buildurunheader.F must also be copied from iolib and they must call respectively the routines buildxxxeve and buildxxxrunheader.
The run is controlled by the RUN (Run number), NEVE (Number of events) and RDEB (Number of events debugged), usually locate in the *rem.cards*.
The file remurun.F can include e.g. event dumping and histogram filling.

5.2 Event generator

An event generator is a REM application without input file with a non trivial routine *generevent*.
The first step for constructing an event generator *gggg* consists in building a IO tool as detailed before.
Furthermore a *bank* directory and a library *src* directory are needed.
In the *bank* directory there are bank descriptor files.
The files *builduevent.F buildurunheader.F generevent.F* in the executable directory must call respectively *buildgggg.F buildggggrunheader.F genergggg.F* in the library directory.
The files *buildgggg.F buildggggrunheader.F* together with *fillgggg.F fillggggrunheader.F* in the library directory must call respectively the *build_* and *fill_* files generated from the banks in the right order to build and fill an event and the run header. The file *genergggg.F* in the library directory must contain the code for filling the event bank(s).
The REM user functions in the executable directory must call the corresponding routines in the library directory, e.g. *remucar* must call *ggggcar*. That allows to event generator to be called as a module by other REM applications.

5.3 Geant simulation

Another important REM based application is the detector simulation with GEANT3.
The first steps are the same that in the previous paragraph on event generator. Nevertheless the presence of the GEANT application that is not built around REM requires some modifications.
The executable directory must contain

- cardset.F Calling GFFGO rather tham FFGO
- gudigi.F GEANT user routine for digitization
- gukine.F GEANT user routine for kinematics
- guout.F GEANT user routine for output
- gustep.F GEANT user routine for step
- gutrev.F GEANT user routine for event initialization

The library directory must contain the driving routines for GEANT and detector modules, plus ggggrun.F, that is

- ggggcar.F
- ggggdefhit.F
- gggggdigi.F
- ggggdraw.F
- ggggend.F
- gggggeom.F
- ggggghis.F
- ggggghit.F
- ggggini.F
- gggginitmed.F
- ggggmate.F
- ggggout.F
- ggggset.F
- ggggtmed.F
- ggggtrev.F
- ggggrun.F

Most of them are simply calling in series the Modules required by the GEANT application.
Exception is *ggggrun.F* that substitutes the grun.F routine from GEANT library and must read as

```
C
      SUBROUTINE GGGGRUN
C! GEANT routine in the event loop
C
      implicit none
```

```
#include "geant321/gcflag.inc"
#include "geant321/gctime.inc"

      CALL TIMEL(TIMINT)

      IEVENT = IEVENT + 1

      CALL GTRIGI

      CALL GTRIG

      CALL GTRIGC

      end
```

The executable directory contains all the REM user functions. In general each routine calls the corresponding application routine, but the necessity of integrating the GEANT calls requires some specific arrangements.
The exceptions are the routines *remuini*, *remuend*, *remuset* that reads as

```
      SUBROUTINE REMUINI
C Initialization routine.
      IMPLICIT NONE
#include "load.inc"
#include "geantsize.inc"

C GEANT ZEBRA memory allocation.
      CALL GZEBRA(NGMEM)

C GEANT initialization.
      CALL GINIT

C Initialize FFREAD card
      CALL CARDINI

C Initialize data structure.
      CALL GZINIT

C Define standard particle data.
      CALL GPART

C Define standard material data.
      CALL GMATE

C Initialize parameters
      CALL GGGGINI

C Initialize tracking media parameters
      CALL GGGGINITMED
```

```
      RETURN
      END

      SUBROUTINE REMUEND
C Termination routine.
      IMPLICIT NONE

      CALL GLAST

      CALL GGGGEND

      CALL PRIREP

      RETURN
      END

      SUBROUTINE REMUSET
C Routine for setting parameters
      IMPLICIT NONE

C Check FFREAD data cards and set run parameters
      CALL GGGGSET

C Setup material and tracking media
      CALL GGGGMATE

C Setup material and tracking media
      CALL GGGGTMED

C Unified geometry.
      CALL GGGGGEOM

C Close geometry definition.
      CALL GGCLOS

C Define the geometrical drawing parameters
      CALL GGGGDRAW

C Define active volumes
      CALL GGGGDEFHIT

C Define cross-sections and energy loss tables.
      CALL GPHYSI

C Print the main parameters.
      CALL GPRINT('MATE',0)
      CALL GPRINT('TMED',0)
      CALL GPRINT('VOLU',0)

      RETURN
```

```
END
```

Appendix

IO files

The formats for defining the IO and histogram files in the cards using the FFREAD package are designed to cope with a plurality of IO situations.
There are six cards governing the IO:

- FILEIN Input file(s)
- DIRIN Input directory
- FILEOUT Output file(s)
- DIROUT Output directory
- FILEHIS Histogram file(s)
- DIRHIS Histogram directory

Each of these cards can be present or not. If FILEOUT or FILEHIS is absent, no output or histogram file respectively is produced. If FILEIN is absent, events will be generated by the routine *generevent*. The default routine in the library iolib returns no event and stop the process. Otherwise a new routine *generevent* must be supplied for creating a new event.
Each of the file cards can supply an absolute file path or a relative one. If the path is relative, the path is completed with the corresponding directory card if it exists, otherwise the current directory is assumed.
The file cards can identify a single file or a parametrized file set. That is done using % as wild card representing a single character. For example FILEIN can be given through

```
FILEIN 'filein_%%%%_%.rz'
FILEPARIN 1000 1 1000 2 1000 3 1001 1 1001 2 1001 3
```

That means that the input file set is parametrized by two parameters, the first with four fields and the second with one, that are read in series from the card FILEPARIN. If necessary padding with 0s will fill the wild card fields.
The output card FILEOUT as well as the histogram flag FILEHIS can be a single file or a parametrized file set with the same number of parameters as FILEPARIN.

References

[1] B. Stroustrup. *The C++ programming language*. Addision Wesley, Reading, MA, 1997.

[2] CERN Program Library Group. FFREAD Format Free Input Processing. CERN Program Library Long Writeups Q123.

[3] R. Brun. Hbook. CERN Program Library Long Writeups Y250.

[4] O. Schaile. Zebra. CERN Program Library Long Writeups Q100/Q101.

[5] P.W. Cattaneo. Input/output code generation from bank documentation and input/output libraries. Available from the author (paolo.cattaneo@pv.infn.it) or from MEG CVS under meg/doc/bank/.