# Data Model for LHC*b* Calorimetry Software

Ivan Belyaev*

February 8, 2000

**Abstract**

Some general and essential features of the calorimetry data model are discussed.

## Contents

**1 Event Data Model** **1**
1.1 General . . . 1
1.2 *Digits* . . . 2
1.3 *Clusters* . . . 4
1.4 Basic Functors . . . 5

**2 Data Flow** **7**

**3 Event Data Access** **8**
3.1 Native $\mathcal{GAUDI}$ way . . . 8
3.2 Advanced method for *CaloDigit** access via *CaloCellID* index . . . 9
3.3 Sequential access to *CaloDigit** . . . 11

**4 Algorithms** **11**
4.1 *CaloDigitizerAlg* . . . 12
4.1.1 *Digitisation Functors* . . . 12
4.2 *CaloCalibratorAlg* . . . 13
4.2.1 *Calibration Functors* . . . 13
4.3 *CaloClusterizatorAlg* . . . 14
4.3.1 *Clusterisation Functors* . . . 14

**5 Examples & Use-cases** **17**

# 1 Event Data Model

## 1.1 General

It was agreed that all calorimetry software should deal Monte Carlo data in the same manner as real data. Technically it can be implemented in a 3 different ways:

- *"No Monte Carlo" approach*: Object for representing Monte Carlo data and object for representing the real data **are the same** objects. Within this approach, the same functions and algorithms are used for Monte Carlo data and for real data.

*E-mail:Ivan.Belyaev@cern.ch

- *Templated approach*: Monte Carlo data object and real data object **are similar** in the sense that they have the same subset of the most important methods. The essential feature of this approach is the wide usage the templated functions and algorithms.

- *Inheritance approach*: Monte Carlo data object **is a kind of** real data object, e.g. Monte Carlo data object inherits from the corresponding real data object. Within this approach the same functions and algorithms for Monte Carlo data and for real data deal only with pointers (or references) to the base classes.

All three approaches fulfil the main mandatory requirement to have the same codes for processing of the real data and for Monte Carlo data.

The disadvantage of the first approach is the complexity of getting of Monte Carlo information, when one needs to get it. All proposed schemes of connection to Monte Carlo truth informations look quite artificial, complicated and unnatural.

The second approach looks as the least suitable for us due to quite complex communications between complicated templated classes. E.g. it almost unavoidable results in the the existence of split structures (Monte Carlo digits and real data digits, Monte Carlo Clusters and real data clusters and so on ) at all levels of the Calorimetry software.

Currently I do not see any serious disadvantages of the third approach. It looks very natural, elegant and powerful. Especially the flexibility of the third approach is amazing. One can easily to switch off this approach and switch on the first approach changing only an extremely small peaces of codes.

The power, beauty and flexibility of the third approach result in our choice.

Currently we are planning to use the same structures/classes for all sub-detectors of LHC*b* Calorimeter system ( Scintillator Pad Detector, Preshower Detector, Electromagnetic and Hadronic Calorimeters). If one day we recognise the necessity of introducing the difference in the data structures, these changes can be easily incorporated into the overall schema including the inheritance and function overloading.

## 1.2 *Digits*

The proposed structure of digits for calorimetry software consists of 4 classes - *class CaloDigit, class MCCaloDigit, class MCCaloDeposit & class MCCaloSummedDeposit*. The most essential feature is the inheritance of *class MCCaloDigit* from *class CaloDigit*.

The physical meaning of *class CaloDigit* is just an energy deposition in a given cell of the calorimeter. Skipping the technical methods and details (constructors, destructors, setters, serialisation, printout, *const*-versions of getters and other technicalities), the simplified version of *class CaloDigit* can be represented as:

```
class CaloDigit: virtual public ContainedObject {
 public:
  /// Retrieve the energy of this digit
  double               e         () const;
  /// Retrieve the identification of
  /// calorimeter cell of this digit
  const   CaloCellID&  cellID    () const;
 private:
  /// Cell identifier
  CaloCellID      m_cellID;
  /// Calibrated energy in MeV
  /// ("the best knowledge of energy")
  double          m_energy;
```

```
};
```

The physical meaning of *class MCCaloDigit* - it is just an energy deposition in a given cell of the calorimeter with a reference to the Monte Carlo truth information. The analogous "simplified" view of *class MCCaloDigit* is presented here:

```
class MCCaloDigit: public CaloDigit
{
 public:
  /// Retrieve energy deposited in active (sensitive) matherial.
  /// (delegation to MCCaloSummedDeposit)
  inline       double                          activeE () const ;
  /// Retrieve the total deposited energy
  /// (delegation to MCCaloSummedDeposit)
  inline       double                          totalE  () const ;
  /// Retrieve container of Monte Carlo deposits
  /// (delegation to MCCaloSummedDeposit)
  inline       SmartRefVector<MCCaloDeposit>& deposits     ()  ;
  /// Retrieve pointer to  Monte Carlo summed deposits
  inline       MCCaloSummedDeposit*           summedDeposit()  ;
 private:
  // deposits
  SmartRef<MCCaloSummedDeposit>   m_summedDeposit;
};
```

Some methods of *class MCCaloDigit* are just delegation to the underlying Monte Carlo structures.

The physical meaning of *class MCCaloSummedDeposit* is just the energy deposited in the active material of the calorimeter (Scintillator plates) corrected taking into account such phenomena as Birk's law, light collection, and other factors, and the total energy deposited an active and absorber material of the calorimeter system and vector of references to the individual energy depositions for this cell from Monte Carlo particles. the individual energy depositions from Monte Carlo particles are represented with *class MCCaloDeposit*. The "simplified view" of *class MCCaloSummedDeposit* and *class MCCaloDeposit* are presented here:

```
class MCCaloSummedDeposit: public ContainedObject
{
 public:
  /// Retrieve energy deposited in active (sensitive) matherial
  /// there is no method to set activeE manually!
  double activeE() const
  /// Retrieve the total deposited energy
  /// there is no method to set totalE manually!
  double totalE () const
  /// Retrieve (reference to) container of Monte Carlo deposits
  inline       SmartRefVector<MCCaloDeposit>& deposits()        ;
 private:
  // Energy deposited in active calorimeter (scintillator) elements
  //   corrected for Birk's law and other corrections
  double      m_activeE;
  // Total deposited energy (including adsorber)
  double      m_totalE ;
  // deposits
  SmartRefVector<MCCaloDeposit>   m_deposits;
```

```
};

class MCCaloDeposit : public ContainedObject
{
 public:
  /// Retrieve the active energy deposited in the
  /// active/sensitive elements (scintillator)
  inline       double            activeE    () const ;
  /// Retrieve the total deposited energy
  inline       double            totalE     () const ;
  /// Retrieve the reference to the MCParticle )
  inline       MCParticle*       particle   ()       ;
 private:
  /// Active energy deposited in active/sensitive material
  double                  m_activeE  ;
  /// Total deposited energy
  double                  m_totalE   ;
  /// Reference to the particle
  SmartRef<MCParticle>    m_particle ;
};
```

### 1.3 *Clusters*

The proposed structure of calorimeter clusters consists of *class CaloCluster*. Since *class CaloCluster* deals only with pointer to the *class CaloDigit*, there is no necessity to get the additional class for Monte Carlo clusters. The same *class CaloCluster* serves for holding the real data and Monte Carlo data.

The "simplified" structure of *class CaloCluster* is presented here:

```
class CaloCluster: public ContainedObject
{
 public:
  ///
  typedef std::vector<CaloCellID>                CellContainer  ;
  typedef unsigned int                           DigitStatus    ;
  typedef unsigned int                           ClusterStatus  ;
  typedef std::pair< SmartRef<CaloDigit> ,
          CaloCluster::DigitStatus  >            CaloDigitPair  ;
  typedef std::vector<CaloCluster::CaloDigitPair> DigitContainer ;
  ///
 public:
  /// Retrieve the energy of the cluster
  inline        double          e   ()                  const ;
  /// Retrieve the x-position of the cluster (barycenter?)
  inline        double          x   () const                  ;
  /// Retrieve the y-position of the cluster (barycenter?)
  inline        double          y   () const                  ;
  /// Retrieve the (symmetric) covariance matrix
  inline        HepSymMatrix&   cov()                         ;
  inline CaloCluster::DigitContainer::size_type size () const ;
  /// Retrieve owned digits
  inline        CaloCluster::DigitContainer&   digits()       ;
  /// Retrieve the current status of the cluster
  inline        ClusterStatus   status     () const          ;
```

```
 private:
  /// Energy of the cluster in MeV
  double          m_energy;
  /// x-position of the cluster in mm
  double          m_xPosition;
  /// y-position of the cluster in mm
  double          m_yPosition;
  /// Covariance matrix
  HepSymMatrix    m_covariance;
  /// digits
  CaloCluster::DigitContainer  m_digits;
  /// Status of the cluster
  CaloCluster::ClusterStatus   m_status;
};
```

Essential feature is that each digits within the cluster is associated with some flag *CaloCluster::DigitStatus*, which is to be used to distinguish cases of privately owned digits or digits shared between several clusters, or even more complicated classification.

## 1.4 Basic Functors

A set of useful *functors* is defined in namespace *CaloDataFunctor* to be used in conjunctions with STL algorithms. They allow us for fins selection, sorting, finding, removing and transformation between basic types of the chosen data model.

```
namespace CaloDataFunctor
{
  ///
  /// Is the "energy" of the object of type TYPE exceed
  /// the threshold value?
  /// templated functor-predicate
  /// TYPE is required to have valid comparison operation with 0,
  /// and  "->e()" method
  ///      (e.g. CaloDigit*,MCCaloDigit*,CaloCluster* and so on...
  /// can be used for any STL algorithm, like std::find_if
  template <class TYPE>
  class Over_Threshold:
  public std::unary_function< TYPE , bool >;

  ///
  /// Is the "energy" of the one object of type TYPE exceed the
  /// "energy" of another object of type TYPE ?
  /// templated functor-predicate
  /// TYPE is required to have valid comparison operation with 0,
  ///  and  "->e()" method
  ///      (e.g. CaloDigit*,MCCaloDigit*,CaloCluster* and so on...
  /// can be used for any STL algorithm,  especially useful for
  /// sorting
  template <class TYPE>
  class Greater_by_Energy:
  public std::binary_function< TYPE , TYPE , bool >;

   ///
  /// Is the "active energy" of the one object of type TYPE exceed
```

```
  /// the "active energy" of another object of type TYPE ?
  /// templated functor-predicate
  /// TYPE is required to have valid comparison operation with 0,
  /// and  "->activeE()" method
  ///      (e.g. MCCaloDigit*)
  /// can be used for any STL algorithm,  especially useful for sorting
  ///
  template <class TYPE>
  class Greater_by_ActiveEnergy:
  public std::binary_function< TYPE , TYPE , bool >;

  ///
  /// Is the "total energy" of the one object of type TYPE exceed
  /// the "total energy" of another object of type TYPE ?
  /// templated functor-predicate
  /// TYPE is required to have valid comparison operation with 0,
  /// and  "->totalE()" method
  ///      (e.g. MCCaloDigit*)
  /// can be used for any STL algorithm,  especially useful for sorting
  template <class TYPE>
  class Greater_by_TotalEnergy:
  public std::binary_function< TYPE , TYPE , bool >;

  ///
  ///  This functor families allows us to "accumulate" (sum)
  ///  different energies for object of type TYPE, TYPE is required
  ///  to have valid comparison with zero and access to a
  ///  corresponding energy
  //
  template <class TYPE>
  class Accumulate_Energy:
  public std::unary_function< TYPE , double >

  template <class TYPE>
  class Accumulate_ActiveEnergy:
  public std::unary_function< TYPE , double >;

  template <class TYPE>
  class Accumulate_TotalEnergy:
  public std::unary_function< TYPE , double >;

};
```

A typical usage of these basic functors one can find here:

```
typedef std::vector<CaloDigit*>   DigitSeq   ;
DigitSeq digit = ... ; // get input data
///
/// 1) get the digit over the threshold
///
DigitSeq digits1;
const double threshold = 10.0 * GeV;

std::remove_copy_if( digits0.begin() , digits0.end()            ,
```

```
                    std::back_inserter( digits1 )              ,
CaloDataFunctor::Over_Threshold<const CaloDigit*> ( threshold ) );
///
/// here  digit1 contains only pointers to digits which
/// energy exceeds 10 GeV
///
/// 2) sort digits which are over threshold
///
std::sort( digits1.begin() , digits1.end() ,
 CaloDataFunctor::Greater_by_Energy<const CaloDigit*>() );
///
/// here digit1 contains a sorted sequence of digits which
/// energy exceeeds 10 GeV
```

# 2 Data Flow

From a formal point of view the complete data flow within calorimeter software can be roughly represented by a following scheme:

*Simulation* At this step a sequence of objects which behaves like *MCCaloDigit** objects[1] is produces.

*Digitisation* At this step a "transformation" of input sequence of *MCCaloDigit** to the output sequence of objects[2], each of them behaves like *CaloDigit** object. An natural place of this step is just after the *Simulation* before writing objects into tape. But since we want currently to investigate in detail all aspects and all details, currently it is foreseen to keep this step before calorimeter reconstruction. In a time, when we have agree with stable digitisation, it naturally goes from begin of *Reconstruction* phase to the end of *Simulation* phase.

*Calibration* The input sequence of objects of type *CaloDigit** is transformed[3]

*Clusterisation* The input sequence of *CaloDigit** objects is transformed into output sequence of *CaloCluster** objects. Internally it is also in a very transparent way can be split into several "sequence-to-sequence" transformation steps[4]

*Seed Finder* At this step an output sorted(optional) sequence of "Seeds" for Clusterisation is produced from input sequence of *CaloDigit** objects. It also can be split into some steps it a natural way

*Cutter* From input sequence of all *CaloDigit** objects it produces the sorted(optional) sequence of *CaloDigit** objects with energy deposition over the certain cell-depended(optional) threshold.

*Maximum Finder* from the sequence of *CaloDigit** objects, it selects the *CaloDigit** which are *local maxima*[5]

[1] It means that it returns either the pointer to *class MCCaloDigit* objects, or to objects, inherited from this *class MCCaloDigit*

[2] Taking into account that *class MCCaloDigit* inherits from *class CaloDigit*, it can be either the same updated sequence, or a new sequence of *MCCaloDigit** objects, or a new sequence of *CaloDigit** objects. The concrete realization is irrelevant for further discussion.

[3] It can be either replaced(updated), or a new sequence of *CaloDigit**(or *MCCaloDigit**) can be produced.

[4] The following sub-division is an illustration only, the real Clusterisation algorithm and it implementation can be quite different, but this schema utilise the quite common features of almost any Clusterisation algorithms

[5] Whatever it means.

*Analyser* From sequence of *CaloDigit**, which are local maxima, produce (taking into account some additional considerations (borders, etc.) sequence of "Seeds"

*Cluster Maker* From input sequence of "Seeds" it produce the output sequence of *CaloCluster** objects. Also can be split into several obvious steps

*Collector* From input sequence of "Seeds" produce the "PreClusters"[6] - just collections of connected *CaloDigit** .

*Calculator* At this step an input sequence of "PreClusters" is transformed[7] into the output sequence of *CaloCluster** objects. A Summed energy, barycenter position and their covariance matrix is calculated (estimated).

*Corrector (optional)* At this step an information in the sequence of *CaloCluster** objects are updated in more sophisticated way - e.g, by applying the S-wave correction, or event with a fitting by shower shape.

*Matching & Particle ID* At this phase the reconstructed *CaloCluster* objects are matched with reconstructed tracks/segments from other subsystems (and with *CaloCluster*s from other parts of calorimeters in th most optimal way. A photon and $\pi^0$ reconstruction and an electron/hadron discrimination is performed.

The first two steps could be in a quite naturally way combined into one step. Probably it is the most effective realization for Monte Carlo mass production. But one should take into account that on the start phase, till the digitisation procedure is not proved to be stable and fixed, the most frequent way is just to (re)-run the *Digitisation* in the beginning of the *Reconstruction/Clusterisation* step.

One can see how nicely the proposed schema of *CaloDigit*, *MCCaloDigit* & *MCCaloDeposit* fits the data flow. We have an unique flexibility within this schema - some "transformations" can be just a "casting", some "transformations" is just "in-place update", and only a minor part of them are to be such nasty and primitive tricks like copying or creating of new objects. But in any case, if due to some external limitation data update will be not possible, the third way is always available, and only a few lined are to be changed to switch off the nice facilities of the proposed approach. In this case where will be no significant advantage with respect to the scheme, proposed by Olivier.

# 3 Event Data Access

Access to the *CaloDigit* and *MCCaloDigit* objects is discussed in this section.

## 3.1 Native $\mathcal{GAUDI}$ way

A straightforward and generic way of accession the data from *Algorithm* is provided by $\mathcal{GAUDI}$ framework via the notion of *class SmartDataPtr*, which provides us with a fast access to the container(*ObjectVector*) of *CaloDigit** objects (*MCCaloDigit** in the case of Monte Carlo):

```
SmartDataPtr<CaloDigitVector>
 SmartRawContainer( eventDataService() ,
        "/Event/Raw/CaloDigitVector");
```

[6] It can be *CaloCluster** object with a properly defined status-word

[7] or updated in-place

```
  if( !SmartRawContainer ){ // we've got the container and here we have
    // an access to the data
  }else{
    // something wrong, or data are unavailable
  }
//
SmartDataPtr<MCCaloDigitVector>
  SmartMCContainer( eventDataService(),
   "/Event/MC/MCCaloDigitVector");
  if( !SmartMCContainer ){
    // we've got the container and here we have
    // an access to the data
    }else{
    // something wrong, or data are unavailable
    }
```

The size of this container is $\sim 10\% \times \#_{\rm Cells}$[8]. This container represents a nice and compact store of pointers to the calorimeter digits. Advantages are obvious but disadvantages are also obvious:

- In the case of Monte Carlo it is still an container of *MCCaloDigit** pointers! (template!), a (trivial) "transformation" to the base class (*CaloDigit**) is required before real usage of Monte Carlo. I have some consultation with Pavel and he states that within our approach we are able to store Monte Carlo data no in the container of type *ObjectVector<MCCaloDigit>* but in the container of the type *ObjectVector<CaloDigit>* and therefore this disadvantage disappears.
- A "sequential" access to the energy deposition for a given cell. Currently we see that it is one of the most frequent question for any implementation of Clusterisation. This question must be answered in the most effective way. Neither ordinary sequential scan[9] no more sophisticated associative or binary scans[10] are not fast enough. Only access by index ("direct access")[11] is acceptable for us.

## 3.2 Advanced method for *CaloDigit** access via *CaloCellID* index

The main statement of almost all our previous discussions was that we definitely need the object which is able to provide a fast and effective access to the data using *CaloCellID* as an index. An *class CaloDigitCollection* was developed to fulfil this requests and in addition to this it was designed to resolve the first problem, mentioned in the previous section - it returns the *CaloDigit** pointer both for "data" and for Monte Carlo.

The simplified view of this constructions are presented here:

```
class CaloDigitCollection: public CaloCollection<CaloDigit*>
{
 public:
  /// constructor - from "address"
  /// (full path in the Transient Store);
```

[8] Taking the average occupancy in the calorimeter at the level of 10%.
[9] Access time $\propto \mathcal{O}(\#_{\rm digits})$
[10] Access time for each of them $\propto \mathcal{O}(\log(\#_{\rm digits}))$
[11] Access time $\propto \mathcal{O}(1)$

```
  CaloDigitCollection( IDataProviderSvc*   dataService       ,
                       const std::string&  address           ,
                       IMessageSvc*        messageSvc    = 0 );
 private:
  // data provider
  IDataProviderSvc*    m_cdc_dataProvider;
};

///
///
/// Base class
///
                                          /* type of content */
template <class CONTENT                                      ,
                                          /* return type     */
  class         RETTYPE   = CONTENT                          ,
                                          /* container type  */
  class         CONTAINER = std::vector<CONTENT>             ,
                                          /* index     type  */
  class         INDEX     = const CaloCellID&                ,
                                          /* functor type    */
  class         FUNCTOR   = std::unary_function<INDEX,RETTYPE&> >
class CaloCollection :  public CONTAINER , public FUNCTOR
{
  ///
  public:
  ///
  typedef    CONTENT              Content    ;
  typedef    RETTYPE              ReturnType ;
  typedef    INDEX                Index      ;
  public:
  // constructor
                                      /* "default" value */
  CaloCollection( Content def                = Content() ,
                  IMessageSvc* messageService = 0         );
  public:
  // CONTAINER
  // access to CONTAINER interfce:
  // access to the content  itself
  // using CaloCellID as an index
  inline       Content       operator[]( Index id )
  // checked access, need to be catched!
  virtual inline       Content       at( Index id )
  /// FUNCTOR!
  /// access to FUNCTOR facilities
  /// please, pay some attention that return type here
  /// CAN BE DIFFERENT from
  /// the return type of operator [] !
  /// this trick is used!!!
  ///
  virtual       ReturnType  operator() ( Index id ) ;
 private:
  // "default value"
  Content                   m_cc_def;
```

```
  // Message Service
  IMessageSvc*              m_cc_messageService;
};
:
```

Example of usage of this construction:

```
CaloDigitCollection digits( eventDataService(),
               "/Event/Raw/CaloDigitVector",
                messageService() );
CaloCellID id = ... ;
CaloDigit* digit  = digits[id] ; // get energy
if(  0 != digit )
{ std::cout << " energy is equal to " << digit->e() << endl; }
```

This construction provides us with fast and effective (access time $\propto \mathcal{O}(1)$) access to the data. An analogous approach (based on concrete implementation of *class CaloCollection*) can be applied to any containers with not trivial access by index, e.g. for geometry implementation.

### 3.3 Sequential access to *CaloDigit**

One should keep in mind that significant part of calorimeter (sub)-algorithms will have a better performance dealing with sequential access to date (e.g. via the native $\mathcal{GAUDI}$ way). Just to simplify this kind of access and to provide the uniform access to *CaloDigit** object an simpler construction *Digit* (defined in namespace *CaloData* is provided. It helps to construct a sequence of *CaloDigit** objects either from native $\mathcal{GAUDI}$ container *ObjectVector<CaloDigit>* or *ObjectVector<MCCaloDigit>* *CaloDigitVector or MCCaloDigitVector*. The usage of this construction is illustrated by following example[12]:

```
///
typedef std::vector<CaloDigit*> DigitSeq;
///
///
///  1) get the data
///
DigitSeq  digits;
CaloData::Digits( eventDataService()           ,
                  "Address in Transient Store " ,
                  std::back_inserter( digits ) ,
                  messageService()             );
///
```

# 4 Algorithms

In this section skeletons and examples of typical *Algorithm*s are presented. The codes itself can be found elsewhere[13].

[12] Example from *$CALOROOT/Calo/Algorithm/CaloClusterizatorAlg.cpp*
[13] All examples come from *$CALOROOT/Calo/Algorithms*

## 4.1 *CaloDigitizerAlg*

The implementation of digitisation algorithm *CaloDigitizerAlg* can be considered as the simplest example of the *Algorithm* within *Calo*-package[14].

This algorithm gets as input the sequence of *MCCaloDigit** objects and provide them with a correct (taking into account the noise, zero suppression and other factors) value of "measured" energy. Input sequence is declared to the algorithm my its full path in the Transient Store[15].

For output sequence we have 2 possibilities:

- One can declare the full path (address) in Transient Store, where the results of digitisation procedure have to be registered.
- If the address of the output sequence is empty, algorithms interprets this information as request for "in-place" update of input sequence.

A several pairs of addresses of input/output sequences can be declared to this algorithm. This is done via *IProperty* interface. This is done externally in *jobOptions.txt* file, e.g.:

```
//
Digitizer.InputOutputSequences  =
  { "/Event/MC/MCEcalDigs0#/Event/MC/MCEcalDigs1" };
Digitizer.InputOutputSequences +=
     { "/Event/MC/MCHcalDigs0" };
//
```

These lines implies that digitisation algorithm gets the object labeled as */Event/MC/MCEcalDigs0*[16] (of the type *MCCaloDigitVector*) and produce the output sequence named *"/Event/MC/MCEcalDigs1"*[17]. Since in the second line the address of the output sequence is absent, it implies that the input sequence will be updated in memory.

Internally *CaloDigitizerAlg* is just a simple skeleton which gets the input data, perform sequential looping over the sequence of *MCCaloDigit**, invokes the real *digitisation functor* for each *MCCaloDigit**, and performs the output operations.

### 4.1.1 *Digitisation Functors*

Currently 3 types of *digitisation functors* are defined and implemented. Probably they cover all possible needs. There is no attempts to describe the noise and all other essential features. But as it seems to me , there will be no any problems with implementation of such features within the current approach. All 3 functors are defined in namespace *CaloDigitizer*[18]:

*Simplest_Digitizer* It performs just a trivial rescaling of the Monte-Carlo "active energy" (*activeE()*) into "energy" (whatever it meant) of the output digit using an constant rescaling factor. The functional form if $e_{\mathrm{output}} = activeE() \times Scale$.

*Smarter_ Digitizer* It applies the same function (or functor) to each "active energy" of the input Monte Carlo digit to produce the "energy" of the output digit. The functional form is: $e_{\mathrm{output}} = f(activeE())$.

[14] Codes can be found in *$CALOROOT/Calo/Algorithms/CaloDigitizerAlg.h* and *$CALOROOT/Calo/Algorithms/CaloDigitizerAlg.cpp*

[15] The default value is *"/Event/MC/MCCalodigitVector"*

[16] The address of the input sequence is written before hash(#) symbol

[17] The address of the output sequence is written after hash(#) symbol

[18] Codes are available in *$CALOROOT/Calo/Digitizer/*

*Clever_Digitizer* Probably the most general form of all possible digitisation methods. It applies the function which is "channel-dependent" to each "active energy" of input Monte Carlo digit to produce "energy" of the output digit. The functional form is $e_{\rm output} = f_{\rm cellID()}(activeE())$.

In principle, it can be imagined that the full "digitisation" is just a result of the collaborative work between all 3 types of functors, e.g. an the first step a simple correction to a visible energy is applied via *Simplest_Digitizer* , then the (probably random, energy-dependent, but channel-independent) noise correction and correction to the finite ADC precision and non-linearity is applied using the *Smarter_Digitizer* and as the last step a some emulation of "hot","dead", and "bad" channels is performed using (channel-dependent) *Clever_Calibrator* .

## 4.2 *CaloCalibratorAlg*

The implementation of calibration algorithm *CaloCalibratorAlg* can be considered as the essential repetition of the concepts described in the previous sub-section example[19].

The only one essential difference between *CaloDigitizatorAlg* and *CaloCalibratorAlg* algorithms is that *CaloDigitizatorAlg* **explicitly** requires an *MCCaloDigit** objects on its own input, while *CaloCalibratorAlg* requires the objects of the type *CaloDigit** an input objects, and therefore it "calibrates" the Monte Carlo and Data in the same manner.

Internally *CaloCalibratorAlg* is just a simple skeleton which gets the input data, perform sequential looping over the sequence of *CaloDigit**, invokes the real *calibration functor* for each *CaloDigit**, and performs the output operations.

### 4.2.1 *Calibration Functors*

Essentially the same types of *functors* defined for digitisation are also defined for calibration.All of them are defined in namespace *CaloCalibrator*[20]:

*Simplest_Calibrator* It performs just a trivial rescaling of the ""energy" (*e()*) on input digit into "energy" of the output digit using an constant rescaling factor. The functional form if $e_{\rm output} = e_{\rm input}() \times Scale$.

*Smarter_Calibrator* It applies the same function (or functor) to each "energy" of the input digit to produce the "energy" of the output digit. The functional form is: $e_{\rm output} = f(e_{\rm input}())$.

*Clever_Calibrator* Probably the most general form of all possible calibration methods. It applies the function which is "channel-dependent" to each "active energy" of input digit to produce "energy" of the output digit. The functional form is $e_{\rm output} = f_{\rm cellID()}(e_{\rm input}())$.

The collaborative work of several such functors also looks quite reasonable, especially if one keeps in ming that usually "calibration" is an iterative procedure. One performs the calibration, then finds a new "constants" or "functions", and again performs the calibration.

[19]Codes can be found in *$CALOROOT/Calo/Algorithms/CaloCalibratorAlg.h* and *$CALOROOT/Calo/Algorithms/CaloCalibratorAlg.cpp*

[20]Codes are available in *$CALOROOT/Calo/Calibrator/* directory.

## 4.3 *CaloClusterizatorAlg*

This is a fist non-trivial algorithm. One can foreseen that the input of this algorithm is the sequence of *CaloDigit** objects and the output is just the sequence of *CaloCluster** objects; As it was implemented for previous *Algorithm*s a several pairs of input/output sequences can be defined using the same facilities in input *jobOptions.txt* file:

```
//
Clusterizator.InputOutputSequences  =
  { "/Event/Raw/EcalDigs#/Event/Rec/EcalClust" };
Clusterizator.InputOutputSequences +=
     { "/Event/Raw/HcalDigs#/Event/Rec/HCalClust" };
//
```

*CaloClusterizatorAlg* has more driving options from *jobOptions.txt* input file. E.g. a some *"prepended" Algorithms* can be forced to be executed as a sub-algorithms of *CaloClusterizatorAlg*. *CaloDigitizerAlg* and *CaloCalibratorAlg* seem to be good candidates for such *"prepended" sub-Algorithms*. Also a some *"appended" Algorithms* can be forced to be executed as a sub-algorithms of *CaloClusterizatorAlg*. A "S-Wave" correction algorithm looks like an excellent candidate for such *"appended" sub-Algorithm*.

### 4.3.1 *Clusterisation Functors*

Currently only 2 "Clusterisation functors" are defined and implemented - SeedFinder. Both functors are defined in *namespace CaloClusterizator* Both functors seek the local maximum. The difference between them is in the type of STL algorithms to be used. The *CaloClusterizator::Is_A_Local_Maximum* functor just selects the digits. which are local maximums. It allows us to use this functor in conjunction with *std::find_if, std::copy_if, std::remove_copy, std::copy* algorithms, while the second functor, *CaloClusterisator::SeedFinder* looks the local maximum and creates the *CaloCluster* objects for found maximums. It is supposed to be used in conjunction with *std::transform* algorithms. Example of usage the latter functor is here:

```
DigitSeq  digits2;
///   4a) create seed finder , at this point we need an object
///       with a fast access to the data using caloCellID as an index
///       and we need the source of geometry info
/// data access in "direct mode"
                           /* data service         */
CaloDigitCollection digitCol ( eventDataService() ,
                           /* address in the store */
                               input              ,
                           /* to report a problems */
                               messageService()   );
///
///   4b) locate the source of geometry info
const std::string calorimeterAddress =
 "/dd/Structure/Calo/ECAL" ;
SmartDataPtr<DeCalorimeter> calo( detDataService()      ,
                                  calorimeterAddress    );
if( !calo )
 {
  log << MSG::FATAL
      << " unable to locate detector information at address= "
```

```
      << calorimeterAddress << endreq;
         return StatusCode::FAILURE;            // RETURN!!!!
  }
///
///
/// 4c) create the seed finder
                        /* source of geometry information */
CaloClusterizator::SeedFinder seedfinder( calo             ,
                        /* random access to digits        */
                                          digitCol         ,
                        /* to report problems             */
                                          messageService() );
///
/// 4d) select the "seeds"
ClusterSeq  preclusters;
std::transform( digits1.begin()                   ,
                digits1.end  ()                   ,
                std::back_inserter( preclusters ) ,
                seedfinder                        );
///
///
/// 4e) remove NULLs
ClusterSeq clusters;
std::remove_copy( preclusters.begin()           ,
                  preclusters.end  ()           ,
                  std::back_inserter(clusters)  ,
                 (const CaloCluster*) 0        );

///
///
///   4f) sort the "seeds" according decreased energy
std::sort( clusters.begin()                              ,
           clusters.end  ()                              ,
 CaloDataFunctor::Greater_by_Energy<const CaloCluster*>() );

///
/// at this point clusters is a sorted container
/// with "clusters" - local maxima
```

The implementation of the functor is so trivial that it it more simpler to list it here than to describe:

```
 ///
 /// The "sophisticated" functor
 ///  if the digit is a local maximum, creates the CaloCluster
 ///  object and fill it
 ///  if the digit is not a local maximum, return NULL pointer
 ///
 class SeedFinder :
 public std::unary_function< const CaloDigit* , CaloCluster*>
   {
   public:
  ///
  /// constructor
     SeedFinder( DeCalorimeter*        det         ,
```

```
              CaloDigitCollection&  digcol      ,
              IMessageSvc*          messageSvc  );
///
/// the only main and essential method
inline CaloCluster*  operator() ( const CaloDigit* digit )
  {
    /// NULL pointer is never local maximum!
    if( 0 == digit ) { return 0; }

   ///
   /// vector of neighbors cell IDs
   const CaloNeighbors* CellIDs =
      &detector->neighborCells( digit->cellID() );

   /// transform neighbour cell IDs  container
   ///           into container of digits
   typedef std::vector<const CaloDigit*> DigSeq;
   DigSeq cells;
   transform_ref ( CellIDs->begin()  , CellIDs->end  ()  ,
                   std::back_inserter( cells )           ,
   // NB - it is an example of usage of
   // "functor" properties of CaloDigitCollection class
                   *digitCollection                     );

   ///
   /// try to find the neighbour with larger energy
   ///
   DigSeq::const_iterator it =
   std::find_if( cells.begin() ,
                 cells.end  () ,
                 std::bind2nd(
 CaloDataFunctor::Greater_by_Energy<const CaloDigit*>(),digit ) );
  ///
  /// this digit is NOT local maximum
  if( cells.end() != it ) { return 0; }
  ///
  /// this digit IS a local maximum!
                ///
       /// this digit IS a local maximum!

       CaloCluster* cluster = new(std::nothrow)  CaloCluster();
       if( 0 == cluster ) { return 0  ; }   //RETURN???

/// add this digit into cluster with status = 1
    cluster->addDigit(
    CaloCluster::CaloDigitPair( digit , 1 ) ); // as an example

/// add all other digits into cluster with status == 2
    DigSeq::const_iterator iter = cells.begin();
    while( cells.end() != iter )
    {
    cluster->addDigit(
    CaloCluster::CaloDigitPair( *iter++ , 2 ) ) ; } // as an example
/// set cluster status
```

```
    cluster->setStatus( 1 ) ;  // as an example
/// return cluster
    return cluster;
}
 private:
   ///
   /// source of detector information
   /// about neighbouring cells
   DeCalorimeter*          detector;
   ///
   /// source of digit information about energy
   /// deposition for a givel cellID;
   CaloDigitCollection*    digitCollection;
 };
```

# 5 Examples & Use-cases

A lot of examples were illustrated in previous sections. In addition the set of examples and frequent "use-cases" is presented here.

- *How one can get the particle with maximum (active) energy deposition in the given cell?*

```
MCCaloDigit* mcdig = ... ; // get digit
/// extract the MCCaloDeposit with
/// maximal deposited active energy
MCCaloDeposit* dep =
*(std::max_element( mcdig->deposits().begin() ,
                    mcdig->deposits().end()   ,
  CaloDataFunctor::Greater_by_ActiveEnergy
              <SmartRef<MCCaloDeposit> >() ));
// extract the Monte Carlo particle
MCParticle* particle = dep->particle;
```

- *What is the total deposited energy in the Calorimeter?*

```
// here     "digits" is a pointer to container of digits
double EnergyInCalorimetry =
 std::accumulate( digits->begin() ,
                  digits->end()   ,
                  0.0             ,
 CaloDataFunctor::Accumulate_Energy<CaloDigit*>() );
```

- *What is the summed energy of all clusters ?*

```
// here    "clusters" is a pointer to container of clusters
double EnergyOfallClusters =
 std::accumulate( clusters->begin() ,
                  clusters->end()   ,
                  0.0             ,
 CaloDataFunctor::Accumulate_Energy<CaloCluster*>() );
```

- *What is the number of digits with energy larger then 10 GeV?*

```
// here     "digits" is a pointer to container of digits
  unsigned long NumberOfDigitsOverThreshold =
     std::count_if( digits->begin() , digits->end(),
CaloDataFunctor::Over_Threshold<const CaloDigit*>(10.0*GeV) );
```

- *What is the number of clusters with energy lower then 50 GeV?*

```
// here    "clusters" is a pointer to container of clusters
  unsigned long NumberOfclustersBelowThreshold =
     std::count_if( clusters->begin() ,
                    clusters->end(),
std::not1(CaloDataFunctor::Over_Threshold<const
                    CaloCluster*>(50.0*GeV) ) );
```

- *What is the number of clusters with less or equal 2 assocoated digits?* They are potential candidates to be identified as MIPs. Since there is no "standard" predicate(functor), we should first define it, and then use:

```
// define predicate(functor)
template <class T>
class size_less_or_equal:
public std::unary_function<T,bool>
{
  unsigned int s;
 public:
  explicit size_less_or_equal( unsigned int i): s(i){};
  inline bool operator() ( const T& x) const
           { return x->size() <= s ; }
};
//
// here    "clusters" is a pointer to container of clusters
unsigned int NumberOfClustersWithLowMultiplicity
 = std::count_if( clusters>begin() , clusters->end() ,
size_less_or_equal<const CaloCluster*>(2) );
\item {\it  What is the mean x-value for
```

- *How to print all valid digit pointers to* `std::cout` *using comma as a delimiter?* It is good illustration of STL algorithms.

```
// here     "digits" is a pointer to container of digits
 std::copy_remove( digits->begin() ,
                   digits->end()   ,
  std::ostream_iterator<std::ostream>(std::cout,","),
           (const CaloDigit*) 0 );
```