Velo Event Model

Internal Note

Issue:	1
Revision:	0
Reference:	LHCb 2006-054
Created:	August 10, 2006
Last modified:	October 4, 2006

Prepared by: Tomasz Szumlak, Chris Parkes (University of Glasgow)

Abstract

This note presents the Velo Event Model, it describes the classes and data flow used in the Velo software for all stages up to and including the clusters. The description is provided for the classes used for both the real data and the simulation. This description includes the data classes used during the standard running and simulation of the experiment, and all classes defined for calibration and commissioning.

Contents

1 Introduction . . . **3**

2 Velo Data Processing- General Description . . . **3**
2.1 Processing of the Real Data . . . 3
2.2 Processing of the Simulated Data . . . 7

3 Velo Event Model Classes . . . **7**
3.1 Data specific classes . . . 7
3.2 Simulation specific classes . . . 10
3.3 Common classes . . . 11

4 Access to Data . . . **15**
4.1 Direct access by key . . . 15
4.2 Sequential access . . . 15

5 Present zero suppressed data processing implemented for DC06 . . . **15**
5.1 Software implementation . . . 16
5.2 Data flow . . . 16
5.3 Associators . . . 16
5.4 Monitors . . . 18

6 Packaging . . . **18**

7 Conclusion . . . **18**

8 References . . . **19**

List of Figures

1 Velo data streams. . . . 3
2 Simplified structure of the RawEvent object, only the Velo related banks are presented . . . 4
3 Structure of the raw bank that is produced as a part of MEP frame . . . 4
4 Conversion of the real data from the MEP frame or binary file into the RawEvent. . . . 5
5 Processing performed on the real data. . . . 5
6 Data flow of the real NZSD. . . . 6
7 The TELL1 emulator - processing stages. TES locations of the produced data are given using italic fonts. . . . 6

8 Processing performed on the simulated data. Produced data types are given (italic fonts). . . . 7
9 Simulated data flow. 8
10 VeloFullBank class (public accessor methods are listed along with constructors). 9
11 ErrorBank class (public accessor methods are listed along with the constructor). 9
12 EvtInfo class (public accessor methods are listed along with the constructor). 10
13 VeloODINBank class (public accessor methods are listed along with the constructor). . . . 11
14 MCVeloFE class (public accessor methods are listed along with thse constructors). 12
15 VeloTELL1Data class (public accessor methods are listed along with the constructor). . . . 13
16 VeloLiteCluster class (public accessor methods are listed along with the constructors). . . . 14
17 VeloCluster class (public accessor methods are listed along with the constructors). 14
18 VeloDigit class (public accessor methods are listed along with the constructors). 16
19 The data flow for the Velo Event Model for DC06. 17

List of Tables

1 Introduction

This note introduces the event model used for the Velo detector. This event model was adopted by the Velo in the period since autumn 2005 when the 1MHz readout scheme was introduced.

The overall data processing and flow in the Velo is presented in section 2. A detailed description of the software implementation of the new Velo Event model is given in section 3. The four different types of data banks used in the output of the Velo TELL1 are reviewed in section 3.1. These banks have a complex structure and are decoded into a set of useful objects for use in the processing and monitoring applications, these objects are described in this section. The objects used only in the simulation of the Velo are presented in section 3.2. Finally, the classes used for both simulated and real data from the TELL1 board (including the standard zero-suppressed cluster) are described in section 3.3. Access to the all data types (decoded real data and simulated data for both non-zero and zero suppressed branches) is presented in section 4. Section 5 contains a short description of the processing chain presently implemented for zero suppressed simulated data as it is used for DC06. Packaging issues and conclusions are given in the sections 6 and 7 respectively.

This note describes only the data formats used, it does not describe the processing algorithms. The TELL1 processing algorithms and the Velo simulation will be the subject of future notes.

2 Velo Data Processing- General Description

The data types related with the Velo are represented schematically in Figure 1. Along with analysis of the real data we need to implement an analysis chain for simulated data that will allow us to understand and debug the Velo hardware during the forthcoming test beam and subsequent detector comissioning. Both real and simulated data consist of two types: non-zero suppressed and zero suppressed. The first type represents the complete data stream coming from the hardware with no thresholds or cuts applied (in other words the data consists of signals - ADC counts - from each active Velo channel) while the second type of data consists of clusters. The following subsections will present in detail issues related with processing performed on the data and the data flow.

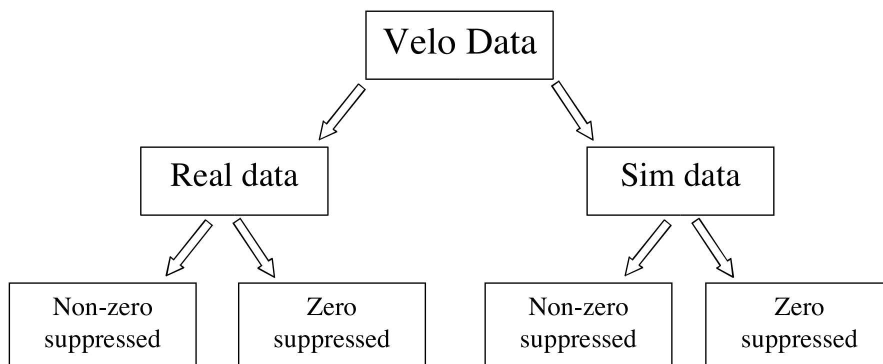

Figure 1 Velo data streams.

2.1 Processing of the Real Data

The output formats of the TELL1 boards are described by the `RawBanks`. The structure of these banks is briefly reviewed in this section. The structure of these banks is complex and they are decoded into a series of other objects which are described in the following sections of this note for use in Gaudi algorithms.

The `RawEvent` contains the collection of the `RawBanks`. The structure of the `RawEvent` is created by dedicated algorithms during the simulation of digitization phase and is identical to that created by the TELL1 [1] electronics, allowing direct comparison between simulation and data. The structure of the `RawEvent` bank for the Velo is illustrated in figure 2. The `RawEvent` is implemented as a map which uses the bank type (`RawBank::BankType`) as the key and provides a vector of pointers to the data banks (`RawBank`).

There are four `RawBanks` that contain the data for the Velo: Velo; VeloFull; VeloPedestal; and VeloError. These bank names are used as the `RawBank::BankType` keys for the `RawEvent` and are defined by an enum

Figure 2 Simplified structure of the RawEvent object, only the Velo related banks are presented

variable inside the `RawBank` class. Each bank contains the data from a single TELL1 (one TELL1 is used per sensor). Hence, in the full Velo setup the vector has 88 pointers to the appropriate data. In the simulation this structure is produced directly by algorithms in the `VeloDAQ` package. In the data multi-event packets (MEP) are used to send the data over the network and the structure is illustrated in Fig.3. Details of the event building procedure preparing are presented in [2] see also [3]. A short description of the raw banks is given below.

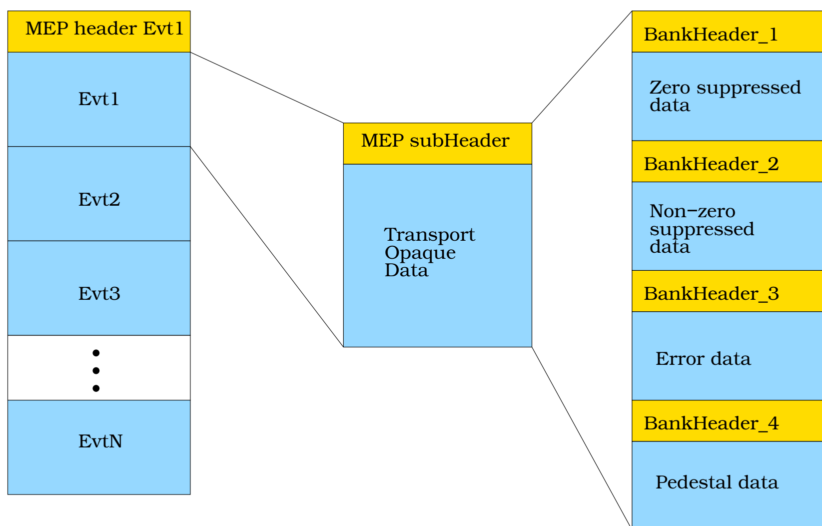

Figure 3 Structure of the raw bank that is produced as a part of MEP frame

Velo This is the standard zero-supressed data format after the TELL1 clustering algorithm. The data format is described in [4]), note that the length of the bank is variable since it is dependent on the number of reconstructed clusters. This data is unpacked into the **VeloCluster** and **VeloLiteCluster** objects.

VeloFull The non-zero supressed data format containing the raw digitised values of all ADC channels. The data format is described in [5] and the bank is of fixed length. The data is unpacked into the **VeloTELL1Data** objects.

VeloPedestal The pedestal values calculated for each strip by the TELL1 pedestal algorithm. The format description is given in details in [7]. The pedestal bank is of fixed length.

VeloError This contains information on synchronisation errors on the TELL1 board and is described in [6]. The error bank is always present for each event and consists of four sections (each of which is related with one PPFPGA processor). In the case when no errors are present only basic default information is sent out for each PPFPGA processor. If synchronisation errors are detected additionally all the Event Info block is sent out for each TELL1 board processor. Hence, the length of the bank is variable.

The first stage of the real data processing consists of conversion of either MEP frame or binary file content (depending on the data acquisition mode), using the dedicated Gaudi converter service, and writing out the

`RawEvent` structure in the TES (see Fig. 4).

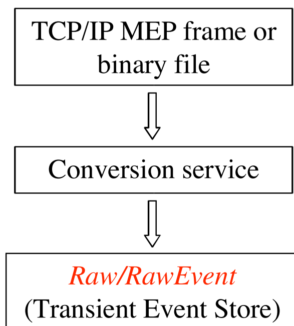

Figure 4 Conversion of the real data from the MEP frame or binary file into the RawEvent.

Having the `RawEvent` inside the event store one can use the specially developed decoding package to unpack the data stored in the `RawBanks`. Processing performed on the data described above is presented in detail in Fig. 5. In the case of zero suppressed data (ZSD) that contains the coded cluster banks only one stage is necessary i.e decoding to VeloClusters and VeloLiteClusters (see Fig. 6).

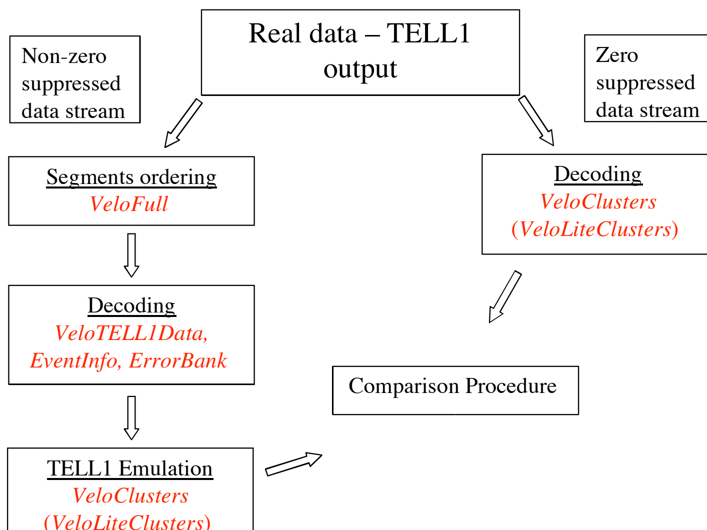

Figure 5 Processing performed on the real data.

The decoded cluster bank can be used in subsequent processing (e.g. data cluster monitoring, Velo standard or generic tracking and data tracking monitoring - the description of these algorithms is outside the scope of this note) depending on user needs. For the non-zero suppressed data (NZSD) the situation is much more complicated. In order to perform analysis of the ADC samples one needs to order the interleaved data from the VeloFull bank and then decode them. After the decoding one can run analog characterization monitoring (so called low level monitoring) which provide essential information about sensor, front-end electronics (Beetle chips) and TELL1 board. To get the cluster bank (which should be identical with the bank decoded from ZSD) TELL1 emulation needs to be run. This is the most complicated stage of the processing and it is performed on both real and simulated NZSD (see 2.2). The TELL1 emulation consists of a number of algorithms that are identical to algorithms implemented in the structure of each Velo TELL1 board's PPFPGA processors. The elements in the TELL1 data processing are: Pedestal following/subtraction; Finite Impulse Response (FIR) digital filtering; bit cutting (reduction from 10 to 8 bit data); data reordering; Linear Common Mode Suppression (LCMS); and clustering. The algorithms use VeloTELL1Data objects for their inputs and outputs. The TELL1 emulation is performed in the Boole or Vetra applications and the output (for data or simulation) is the standard zero-suppressed Velo data format containing the reconstructed clusters. Comparison of the cluster banks, the one obtained from decoding of the ZSD and the one that is an output of the emulator, will make a good base for testing of the TELL1 algorithms performance.

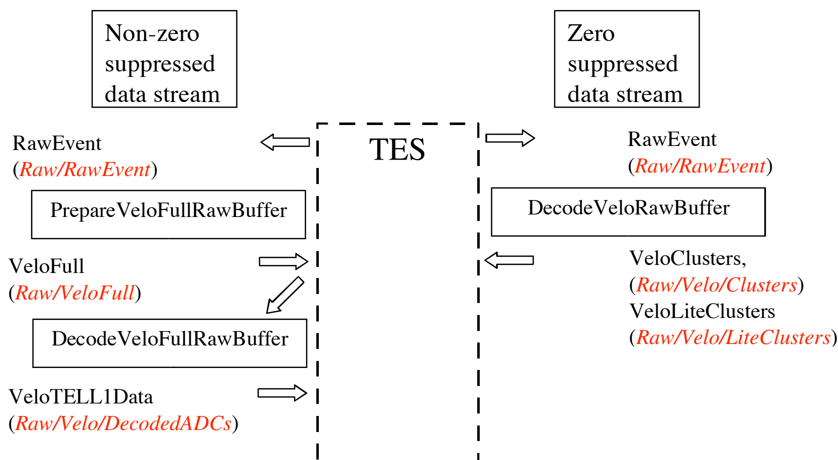

Figure 6 Data flow of the real NZSD.

The TELL1 emulator is currently implemented within the Vetra project as a component. The processing chain performed by the emulator is depicted on Fig. 7 (for each stage of processing TES locations of the input and output data containers are given).

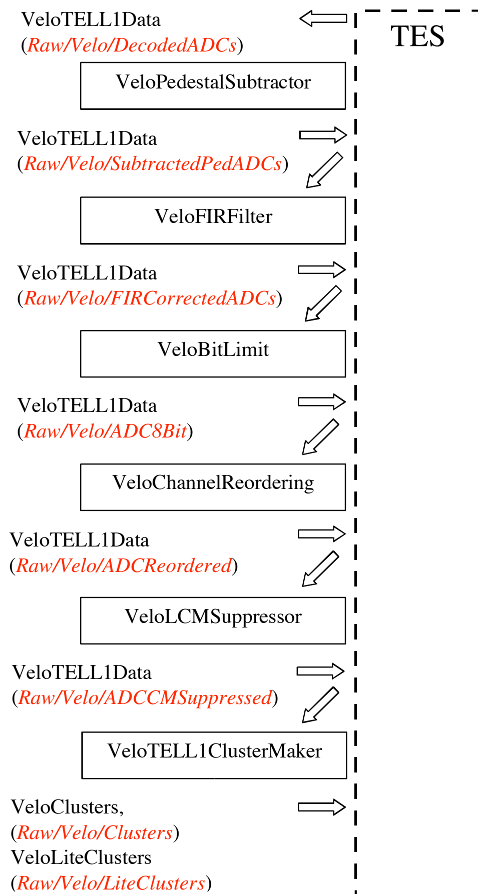

Figure 7 The TELL1 emulator - processing stages. TES locations of the produced data are given using italic fonts.

2.2 Processing of the Simulated Data

The Geant simulation is performed in the Gauss application resulting in the MCHit and MCParticle objects. The silicon and front-end chip simulation is then performed by the VeloSimulation package in the Boole application producing the MCVeloFE objects (in the case of the full Velo setup we obtain about 2000 MCVeloFE objects on average). An emulation of the digitization analog detector signals is performed. This result in the VeloDigits object container. Pedestals and Common Mode noise are not normally simulated (see Fig. 8) and the data volume is kept small by including only channels above a signal threshold cut.

It is also possible to run the silicon smulation in a special non-zero suppressed mode which is presented in Fig. 8. It should be stressed that the input file (with MCHits and MCParticles objects) is the same as in the case of the simulation of the ZSD. The output of the silicon simulation running in this mode is a container of MCVeloFE objects for each channel of the Velo detector (182 000 entries). The data is ordered into chip channel order. In the next step digitization of the simulated analog signals is performed and VeloTELL1Data objects are produced with raw ADC samples. From this stage onwards both real and simulated NZSD are treated in the same way. The last step of the NZSD analysis is the TELL1 emulation procedure that was described in the previous section (the only difference is that the input container for the TELL1 emulator, in the case of simulated data, has the location `Raw/Velo/SimulatedADC`). The output of this stage is again production of the cluster bank.

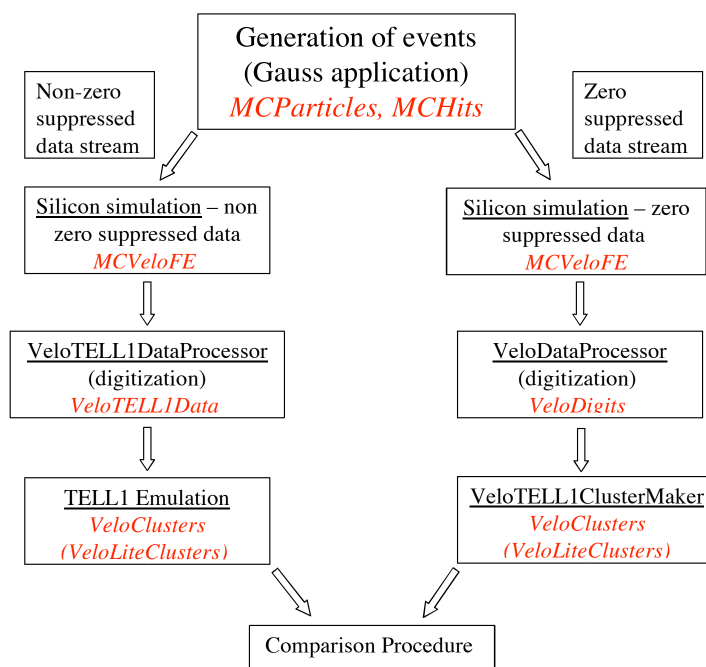

Figure 8 Processing performed on the simulated data. Produced data types are given (italic fonts).

Detailed simulated data flow, corresponding to the processing chains described above, is presented in Fig. 9.

3 Velo Event Model Classes

In the section software implementation of the Velo Event Model will be provided. The description is divided into three independent parts that covers issues related with data specific classes, simulation specific classes and common objects that are used for both data types.

3.1 Data specific classes

A number of data objects are used for commissioning and monitoring the Velo system performance, these include general event information and synchonisation error information. In order to use the information inside Gaudi applications dedicated data classes (inherited from standard Gaudi `KeyedObject` class) have been provided.

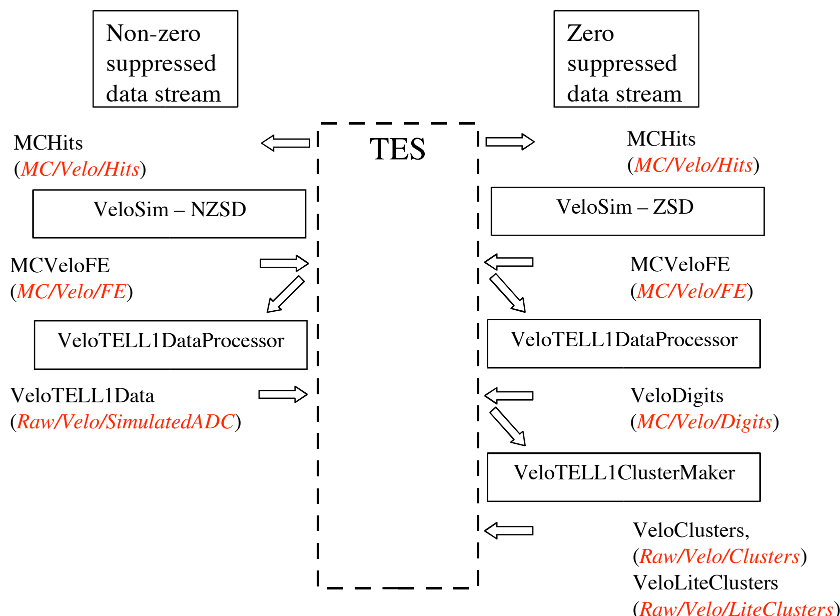

Figure 9 Simulated data flow.

<u>**VeloFullBank**</u>

Short description: Stores coded data ordered into analog links order (where analog link represent 32 physical Velo channels, an analog link is the atomic data unit used in TELL1 processing)

Corresponding Algorithm: Produced by the PrepareVeloFullRawBuffer in the Velo/VeloTELL1Decoding package from RawEvent::VeloFull

Key: The class inherits from the KeyedObject class and uses the sensor number as the key.

TES location:

- Raw/Velo/ADCBank
- Raw/Velo/PedBank

Package: Velo/VeloTELL1Event

Description: The class is used to store non-zero suppressed coded (32 bit words) digitized data and pedestals. In order to create objects of type VeloFullBank one needs to specify as follow: the number of TELL1 boards (sensors) from which the data was sent out, pointer to the beginning of the RawBank data body (unsigned int*) and the data type. The data type variable is needed for proper initializtion and makes it possible to store also data from the pedestal bank. The VeloFullBank objects are the output stream of the PrepareVeloFullRawBank algorithm and are completely decoupled from the simulated analysis chain (see Fig.10).

<u>**ErrorBank**</u>

Short Description: Stores the information on synchronisation errors.

Corresponding Algorithm: Produced by the DecodeVeloFullRawBuffer algorithm in the Velo/VeloTELL1Decoding package from the RawEvent::VeloError.

Key: The class inherits from the KeyedObject class and uses the sensor number as the key.

TES location: Raw/Velo/ErrorBank

Package: Velo/VeloTELL1Event

Description: The class is used to store information about synchronisation errors that could appear during the TELL1 processing. The structure of the `RawEvent::ErrorBank` is very complicated [6]. For each TELL1 board one single error bank is produced and contains four sections (one per PPFPGA processor). To create ErrorBank objects one needs to specify the number of TELL1s and pass it to the object constructor. To get the

```
VeloFullBank : KeyedObject<int>
---
VeloFullBank(const int numberOfTELL1, unsigned int* inBank, const int type)
---
typedef std::vector<unsigned int> section
dataVec& getSection(int block, int section) const   // returns ordered data form bank body as 32 bit int
allEvt& getEvtInfo( ) const                         // returns ordered Event Info block as 32 bit int
---
std::vector<section> m_sections                     // vector with all sections from data bank
unsigned int* m_bank                                // pointer to the beginning of data bank body
std::vector<PPFPGABlock> m_PPFPGABlocks             // vector of PPFPGA data blocks
int m_numberOfWordsInBlock                          // number of 32 bit word in one data block
int m_numberOfWordInSection                         // number of 32 bit words in data section
allEvt m_infos                                      // vector of Event Info block as 32 bit words
```

Figure 10 VeloFullBank class (public accessor methods are listed along with constructors).

information from the stored bank one needs to use a set of accessors that are implemented inside the ErrorBank class (see Fig. 11). This data type is present only in the real zero suppressed data coming from the TELL1 board. The ErrorBank can be stored as empty or full (depending on whether an error ocurred or not). In the latter case each stored ErrorBank contains an EvtInfo object (see below).

```
ErrorBank : KeyedObject<int>
---
ErrorBank(const int numberOfTELL1)
---
unsigned int eventInformation(const int PPFPGA=0) const
unsigned int bankList(const int PPFPGA=0) const
unsigned int detectorID(const int PPFPGA=0) const
unsigned int bunchCounter(const int PPFPGA=0) const
unsigned int clusterDataSectionLenght(const int PPFPGA=0) const
unsigned int adcDataSectionLenght(const int PPFPGA=0) const
unsigned int processInfo(const int PPFPGA=0) const
unsigned int PCN(const int PPFPGA=0) const
unsigned int numberOfCluster(const int PPFPGA=0) const
unsigned int errorBankLenght(const int PPFPGA=0) const
unsigned int nonZeroSuppressedBankLenght(const int PPFPGA=0) const
unsigned int pedestalBankLenght(const int PPFPGA=0) const
unsigned int checkBanks(const int PPFPGA=0) const
bool isEmpty( ) const                                  // check if any errors were sent
std::vector<unsigned int>& errorSources( ) const       // numbers of PPFPGA for which errors were se
EvtInfo* evtInfo( ) const                              // return EvenInfo block
---
EvtInfo m_evtInfoData                                  // Event Info block
allError m_errorInfoData                               // default information from error bank
dataVec m_errorSources                                 // numbers of PPFPGA forwhich error occured
```

Figure 11 ErrorBank class (public accessor methods are listed along with the constructor).

EvtInfo

Short Description:

Stores information from the Event Info block, which is a part of the VeloFull non-zero suppressed data raw bank or is sent out inside the VeloError raw bank in the case of zero suppressed data

Corresponding Algorithm:

Produced by the DecodeVeloFullRawBuffer algorithm in the Velo/VeloTELL1Decoding package from the Velo-Full data for NZSD analysis or from the VeloError raw bank for ZSD

Key: The class inherits from the KeyedObject class and uses the sensor number as the key.

TES location: Raw/Velo/EvtInfo

Description:

Each TELL1 sends four EventInfo sections (one for each PPFPGA processor), a single EvtInfo object stores all four Event Info sections. The information stored inside the EvtInfo are vital for testing and debugging the Velo hardware. Along with some general information (such as the detector identification number, bunch counter or TELL1 number) one can also find information that is specific for the actual event such as L0 event identification number, processing list and bank list. Special information about the performance of the front-end chips is also available (for details see Fig. 12).

```
EvtInfo : KeyedObject<int>
--------------------------------------------------------------
EvtInfo(const int numberOfTELL1)
--------------------------------------------------------------
void setEvtInfo(allEvt& inData)
unsigned int bunchCounter(const int PPFPGA=0) const
unsigned int detectorID(const int PPFPGA=0) const
unsigned int bankList(const int PPFPGA=0) const
unsigned int eventInformation(const int PPFPGA=0) const
unsigned int l0EventID(const int PPFPGA=0) const
unsigned int FEMPCN(const int PPFPGA=0) const
unsigned int processInfo(const int PPFPGA=0) const
unsigned int chipAddress(const int PPFPGA=0) const
unsigned int channelError(const int PPFPGA=0) const
unsigned int adcFIFOError(const int PPFPGA=0) const
unsigned int headerPseudoErrorFlag(const int PPFPGA=0) const
unsigned int PCNError(const int PPFPGA=0) const
unsigned int FEMPCNG(const int PPFPGA=0) const
dataVec IHeader(const int PPFPGA=0) const
dataVec PCNBeetle(const int PPFPGA=0) const
--------------------------------------------------------------
allEvt m_evtInfos              // vector with all event info blocks
```

Figure 12 EvtInfo class (public accessor methods are listed along with the constructor).

VeloODINBank

Short Description:

Stores global information about the event recorded by the ODIN readout supervisor.

Corresponding Algorithm:

The VeloODINBank object is created by the DecodeVeloFullRawBuffer algorithm (for each event exactly one such object is created).

Key: The class doesn't have any key since for each event only one object of the type VeloODINBank is needed. The class inherits from ContainedObject class.

TES location: Raw/Velo/ODINBank

Description:

One of many tasks of the ODIN is to provide some detector specific information (recorded locally) for the DAQ system. In order to do so an ODIN raw bank is created on an event-by-event basis. The bank is subsequently merged with the rest of the event during the event building procedure after the L0 trigger accept signal. Detailed description of this procedure and the ODIN bank content can be found in [10]. Implementation of the class within Gaudi framework is presented in Fig. 13.

3.2 Simulation specific classes

The simulation of the Velo models the response of the silicon and front-end chip to the passage of particles passing through the sensor. The simulation uses the event model classes described below which contain monte-carlo information. The remaining processing for the simulation proceeds as for the data, as described in the other section of this note. The link between the reconstructed objects and the simulation objects is provided via associators which are described in section 4.

The Gauss simulation application's output consists of MCParticle and MCHit objects. The classes are not Velo specific so only a brief description is provided below.

- **MCParticle** This object is used LHCb wide. It is produced in the simulation application Gauss and provides the monte-carlo information on the generated particles.
- **MCHit** This object is used LHCb wide. It describes the entrance and exit points of particles passing through the Velo silicon detectors and the charge deposited in the sensors. The object has a dedicated data member that is used to identify the Velo sensor in which the hit is located.

```
VeloODINBank : ContainedObject
------------------------------
VeloODINBank( )
------------------------------
void setODINBody(dataVec inData)
void setODINSize(unsigned int inValue)
unsigned int odinSize( )
void setODINSourceID(unsigned int in Value)
unsigned int odinSourceID( )
void setODINVersion(unsigned int inValue)
unsigned int odinVersion( )
void setODINType(unsigned int inValue)
unsigned int odinType( )
unsigned int runNumber( )
unsigned int eventType( )
unsigned int orbitNumber( )
unsigned long long L0EventID( )
unsigned long long GPSTime( )
unsigned short int bunchCurrent( )
unsigned short int errorBits( )
unsigned short int detectorStatus( )
unsigned short int bunchID( )
unsigned short int BXType( )
unsigned short int forceBit( )
unsigned short int readoutType( )
unsigned short int triggerType( )
------------------------------
dataVec m_odinBody              // ODIN bank data body
unsigned int m_odinSize         // size of data + size of header
unsigned int m_odinSourceID     // number of ODIN board (set by ECS)
unsigned int m_odinVersion
unsigned int m_odinType         // used by offline to recognize bank type
```

Figure 13 VeloODINBank class (public accessor methods are listed along with the constructor).

MCVeloFE

Short Description: Stores information of simulated analog response of the front-end Beetle chip

Corresponding Algorithm: Produced by the VeloSim algorithm in the Velo/VeloSimulation package using MCHits as input

Key: The class inherits from the KeyedObject class and uses the VeloChannelID as the key

TES location: MC/Velo/FE

Package: Event/MCEvent

Description: These objects are created by the VeloSimulation package that simulates the silicon sensor and front-end chip response to particles passing across the sensor and is completely decoupled from the real data stream. Each MCVeloFE represents a single Beetle chip channel providing the total measured charge (in electrons). The public member functions are given in Fig.14.

3.3 Common classes

There are a number of common classes used to store both simulated and real data. Such unification of the data objects is required because from a certain point of the analysis we want to process the data in the same way.

VeloTELL1Data

Short Description:

Stores the non-zero suppressed data from the TELL1. Stores this data after the various processing stages of the TELL1 emulation are performed. Can also be used to store Pedestal or ADC header data.

Corresponding Algorithm:

Produced by the DecodeVeloFullRawBuffer algorithm in the Velo/VeloTELL1Decoding package from the VeloFullBank objects. For the simulation the object is produced by the VeloTELL1DataProcessor algorithm in the package Velo/VeloTELL1Algorithms. It is produced by the various TELL1 processings algorithm emulations.

```
MCVeloFE : KeyedObject<VeloChannelID>
----------------------------------------------------------------------------------------
MCVeloFE(const VeloChannelID&)
MCVeloFE( )
----------------------------------------------------------------------------------------
VeloChannelID channelID( ) const
long sensor( ) const                                // sensor number
long strip ( ) const                                // strip number
void addToMCHit(const SmartRef<MCHit>&, double)     // add MCHit and its deposited charge
SmartRef<MCHit> mcHit(long) const                   // retrieve reference to MCHit
double mcHitCharge(long) const                      // charge deposited by MCHit
void setMCHitCharge(long, double)
long numberOfMCHits( ) const                        // number of associated MCHits
double charge( ) const                              // total charge in electrons
double addedSignal( ) const                         // signal on strip in electrons
void setAddedSignal(double)
double addedPedestal( ) const                       // pedestal in electrons
void setAddedPedestal(double)
double addedNoise( ) const                          // noise in electrons
void setAddedNoise(double)
double addedCMNoise( ) const                        // Common Mode noise in electrons
void setAddedCMNoise(double)
const std::vector<double>& mcHitsCharge( ) const    // vector with charge of contributed MCHits
void setMCHitsCharge(const std::vector<double>&)
const SmartRefVecotr<MCHit>& mcHits( ) const        // vector of references to MCHits contributed
void setMCHits(const SmartRefVector<MCHit>&)
void addToMCHits(const SmartRef<MCHit>&)            // add reference to MCHit contributing to the channel
void addToMCHits(const MCHit*)
void removeFromMCHits(const SmartRef<MCHit>&)       // remove from the reference vector of MCHits
void clearMCHits( )                                 // clear the vector of MCHits
----------------------------------------------------------------------------------------
double m_addedSignal                       // signal from the channel in electrons
double m_addedPedestal                     // added pedestal in units of electrons
double m_addedNoise                        // added noise in electrons
double m_addedCMNoise                      // added CM noise in units of electron
std::vector<double> m_MCHitsCharge         // charge of MCHits contributing to this channel
SmartRefVector<MCHit> m_MCHits             // references to MCHits contribiting to the channel
```

Figure 14 MCVeloFE class (public accessor methods are listed along with thse constructors).

When storing pedestal and ADC header data these objects are also produced by the DecodeVeloFullRawBuffer algorithm.

Key:

The class inherits from the KeyedObject class and uses the sensor number as the key.

TES:

Since this object is used at different stages of the processing, e.g. for the non-zero suppressed raw ADC data, or after the pedestal algorithm, the created objects are stored at different TES locations after each stage of the algorithm.

These locations are:

- Raw/Velo/DecodedADC
- Raw/Velo/DecodedPed
- Raw/Velo/DecodedHeaders
- Raw/Velo/SimulatedADC
- Raw/Velo/SimulatedPed
- Raw/Velo/SubtractedPed
- Raw/Velo/SubtractedPedADC
- Raw/Velo/FIRCorrected
- Raw/Velo/ADC8Bit
- Raw/Velo/ADCReordered

- Raw/Velo/ADCCMSuppressed

Description:

The object can store real or simulated data for use in the algorithms that emulate the TELL1 preprocessing. The data is stored as 10 bit signed integers, although in the later stages of the FPGA processing only 8 bits are used. This TELL1 emulation chain allows the production of zero-suppressed data from non-zero suppressed data, and allows the cross-checking of the TELL1 algorithms. The object is designed to best interface with the TELL1 algorithms requirements and handles the decoded data as sets of 32 channels. In addition to storing the ADC data (and this data at different stages of the TELL1 processing) this object can also be used to store the pedestal data. This makes it easy to perform a direct comparision of the pedestal algorithm emulation with the TELL1. Furthermore, this object can also be used for the ADC header information that is used during the hardware tests and Beetle channels cross talk studies. The three types of data that can be stored (ADC, Pedestals, ADC headers) are controlled by a type variable passed to the constructor.

A full description of the object is provided in Fig. 15.

VeloTELL1Data : KeyedObject<int>	
VeloTELL1Data(const int numberOfTELL1, const int type)	
void setDecodedData(dataVec& inVec) dataVec& operator[ ](const int ALink) const	// set decoded data // returns 32 decoded objects for one analog link

dataVec m_decodedData
dataVec m_aLink
int m_dataType
	// vector of decoded data (10 bit int words)
	// data for one analog link
	// type of decoded data

Figure 15 VeloTELL1Data class (public accessor methods are listed along with the constructor).

Clusters

In the standard running mode of the Velo the data processing and clustering is performed on the TELL1 board and the zero-suppressed data written out in the `RawBank::Velo` format. In the simulation the output of the Boole application is also stored in the `RawBank::Velo` format. Two cluster classes are provided into which the `RawBank::Velo` data is decoded for use in Gaudi algorithms (i) `VeloLiteCluster` for use in on-line applications and (ii) `VeloCluster` providing full ADC information. In addition a `FastClusterClass` is provided to store the `VeloLiteCluster` rather than using the standard LHCb `KeyedContainer` object. The clusters use the VeloChannelID class to store the strip and sensor number.

VeloLiteCluster

Short Description: Clusters for use in fast online algorithms.

Package: Event/DigiEvent

Corresponding Algorithm: The `VeloLiteCluster`s are decoded from the RawBank::Velo by the `DecodeVeloRawBuffer` algorithm in the `Velo/VeloDAQ` package.

Key: It is not a keyed object. The VeloLiteCluster is stored in a FastClusterContainer class on the TES. Although VeloLiteCluster is not a KeyedObject it is possible to retrieve the clsuters from the container using the VeloChannelID of the strip closest to the cluster centre.

TES location: Raw/Velo/LiteClusters

Additional Information: The VeloLiteCluster has only limited information about cluster position (3 bit precision on the weighted strip centre) and cluster size (one can get accurate information of cluster size if the cluster has one or two strips all clusters with 3 strips or more are not distinguishable) and no ADC information is available. The design of this object is driven by the requirements of the L1 trigger. In order to minimise the size of the object there are no virtual methods defined inside the class (in order that we don't have a virtual table). There gives a gain in speed since we don't need to perform additional dereferencing for virtual functions. The implementation of this class is common with the ST. The class is presented in figure 16.

VeloCluster

Short Description: Clusters for use in offline algorithms.

```
VeloLiteCluster
-------------------------------------------------------------------------------------------
VeloLiteCluster(const VeloChannelID&, double fracStrip, unsigned int size, bool secondThres)
VeloLiteCluster(unsigned int fracStrip, unsigned int pseudoSize, bool second Thres,
                const VeloChannelID&)
VeloLiteCLuster( )
-------------------------------------------------------------------------------------------
VeloLiteCluster& operator=(const VeloLiteCluster&)  // assignment operator
double interStripFraction( ) const                  // position of the cluster centre
unsigned int pseudoSize( ) const                    // size of the cluster (3 means 4 strips or more)
VeloCahnnelID channelID( ) const                    // channel ID of the cluster
bool isRType( ) const
bool isPhiType ( ) const
bool isPileUp( ) const
bool highThreshold( ) const                         // retrieve the high threshold
-------------------------------------------------------------------------------------------
emun liteClusterBits                                // bit structure of lite cluster
enum liteClusterMasks                               // masks of lite cluster
unsigned int m_liteCluster                          // lite cluster
```

Figure 16 VeloLiteCluster class (public accessor methods are listed along with the constructors).

Corresponding Algorithm: The `VeloClusters` are decoded from the RawBank::Velo by the `DecodeVeloRawBuffer` algorithm in the `Velo/VeloDAQ` package.

Package: Event/DigiEvent

Key: A KeyedObject with the **VeloChannelID** object as key.

Description: The VeloCluster object is used by the HLT/off-line tracking. The objects contains all the information in a VeloLiteCluster (VeloLiteCluster is a data member of the VeloCluster). In addition it is possible to retrieve the ADC counts (after pedestal subtraction and common mode suppression) for all strips contributing to the cluster object as well as the strip's VeloChannelID. This information can be used to determine a more accurate position of the centre of a given cluster for use in the offline processing. In order to retrieve this information one can get the ChannelID of the strip having the highest ADC count amongst all those strips contributing to the given cluster and combine that information with the inter strip position. The cluster centre is calculated in terms of the pitch distance between strips using the VeloClusterTool in the Velo/VeloTools package. The tool also computes the corresponding error (in units of pitch distance as in the case of the inter strip distance). The class is presented in figure 17.

```
VeloCluster : KeyedObject<VeloChannelID>
-------------------------------------------------------------------------------------------
VeloCluster(const VeloLiteCluster&, const ADCVector&)
VeloCluster( )
-------------------------------------------------------------------------------------------
unsigned int size( ) const                        // number of strips in cluster
unsigned int adcValue(unsigned int) const         // adc value for a given strip
int strip(unsigned int) const                     // strip number of strip in cluster
VeloChannelID channelID( ) const                  // channel ID of the LiteCluster
double totalCharge( ) const                       // charge summed over all strips in cluster
double interStripFraction( ) const                // position of the cluster centre
unsigned int pseudoSize( ) const                  // size of the cluster (max is 3)
bool highThreshold( ) const                       // set if cluster's adc count is above cut
VeloChannelID firstChannel( ) const               // channel ID of the first strip
VeloChannelID lastChannel( ) const                // channel ID of the last strip
std::vector<VeloChannelID> channels( ) const      // channel IDs for all strips
bool isRType( ) const
bool isPhiType( ) const
bool isPileUp( ) const
const ADCVector& stripValues( ) const             // strips numbers and their signals
void setStripValues(const ADCVector&) const       // set the stip numbers and their signals
-------------------------------------------------------------------------------------------
VeloLiteCluster m_liteCluster     // TELL1 cluster without ADC values
ADCVector m_stripValues           // vector with strip numbers and their ADC signals
```

Figure 17 VeloCluster class (public accessor methods are listed along with the constructors).

4 Access to Data

All the data classes described above are of the type KeyedObject so it is possible to get access to the data in sequence using iterators or to a single object using its key. Below a detailed description of the VeloTELL1Data is presented all other objects could be accessed in a similar manner.

4.1 Direct access by key

Let us consider that one wants to access the NZS data coming from a specific TELL1 board. The first step will be to retrieve data from the TES:

```
// check the availability of the data and get it
if(!exist<VeloTELL1Datas>(VeloTELL1DataLocation::ADCs))
 info()<< " ==> There is no decoded ADCs at: "
      << VeloTELL1DataLocation::ADCs <<endmsg;
 return (StatusCode::FAILURE);
}else{
// get the data from specified location
m_decodedADCs=get<VeloTELL1Datas>(VeloTELL1DataLocation::ADCs)
}
```

Having the container one can ask about an object with a given key (in this case the number of the required TELL1)

```
// desired TELL1
const int TELL1No=23;
VeloTELL1Data* dataFromSensor;
dataFromSensor=m_decodedADCs->object(TELL1No);
//get the data for first analog link (32 channels)
std::vector<signed int> analogLink;
analogLink=(*dataFromSensor)[0];
// do something with the data ...
```

4.2 Sequential access

Let us now consider looping over the whole container of VeloTELL1Data objects (this time we skip retrieving the data from the TES, since the procedure remains the same for both tyes of access)

```
// iterator to VeloTELL1Data objects
VeloTELL1Datas::const_iterator sensIt=m_decodedADCs->begin();
// loop over the senors
for( ; sensIt!=m_decodedADCs->end(); ++sensIt){
 VeloTELL1Data* dataFromSensor=(*sensIt);
 std::vector<signed int> analogLink;
 for(int ALink=0; ALink<NumberOfAlinks; ++ALink){
   analogLink=(*dataFromSensor)[ALink];
   // do something useful with the data ...
 }
}
```

5 Present zero suppressed data processing implemented for DC06

The data model used for DC06 is essentially the same as that presented in the rest of this note. However at this time the TELL1 algorithm emulation was not available. Hence two further internal classes were used to account

for this. These two classes are presented below for completness but are only of interest for internal Velo studies. The stored DC06 data is fully compatible with the final event model presented in this note.

5.1 Software implementation

In order to provide suitable Velo Event Model for DC06 following classes have been defined: MCVeloFE, VeloDigit, InternalVeloCluster, VeloCLuster and VeloLiteCluster. Below we give description of the VeloDIgit and InternalVeloCluster only since all other classes have been described in previous section.

- **VeloDigit** the class inherits from KeyedObject with VeloChannelID as a key (Fig.18). An object of type VeloDigit stores a single raw ADC sample. It is also possible to retrieve the information provided by using the VeloChannelID (strip, sensor). The container to store the VeloDigit objects is produced by the VeloDataProcessor and used to produce clusters.
- **InternalVeloCluster** inherits from KeyedObject also with VeloChannelID as the key. An object of type InternalVeloCluster contains all the information of contributing strips and their ADC signals. All VeloChannelIDs are also available. The clusters are produced by the VeloClusterMaker algorithm and are used as an input for the PrepareVeloRawBuffer algorithm that initializes and fills up the zero suppressed bank in raw event. To get VeloLiteClusters and VeloClusters one needs to perform decoding sequence (DecodeRawBankToLiteClusters and DecodeRawBankToClusters routines).

VeloDigit : KeyedObject<VeloChannelID>	
VeloDigit(const VeloChannelID& key) VeloDigit( )	

VeloChannelID channelID( ) const
long sensor( ) const
long strip( ) const
short int adcValue( ) const
void setADCValue(short int value)
	// get access to channel info
	// sensor number
	// strip number
	// value of ADC sample

short int m_ADCValue	// ADC sample

Figure 18 VeloDigit class (public accessor methods are listed along with the constructors).

5.2 Data flow

The data flow for the Velo Event Model for DC06 is shown on Fig.19. The algorithms sequence presented here allows the production of simulated zero suppressed like data even in the absence of the final TELL1 algorithms. Starting from the MCHits container that is given by the event generation program we produce the MCVeloFEs container. Next the VeloDigits container is created with ADC samples. A number of cuts are applied to the VeloDigit objects so they should be treated as zero suppressed data. One container is produced for all read out sensors. The digitized samples are then passed to the clusterization algorithm and InternalVeloCluster objects are created. Neither the clusters nor the clusterization algorithm are the final TELL1 version therefore the clusters can't be put directly into the raw bank. Hence an intermediate algorithm is needed to perform the coding of the internal clusters to the raw buffer zero suppressed VeloCluster objects.

5.3 Associators

Associators are algorithms that provide links between different objects defined within the data model. For the DC06 new associators have been created using linker classes [9]. It has been decided to create two association tables with links between VeloClusters and MCParticles and between VeloDigits and MCParticles. To create these associations we used VeloChannelIDs (keys of the associated objects) and in addition weights have been defined accordingly. The weights are related with the charge deposited by the MC object contributing to the reconstructed one. The main advantage coming from using linker object is that there is one simple and unified interface. For example one can retrieve a table with associations from the VeloDigits to the MCHits:

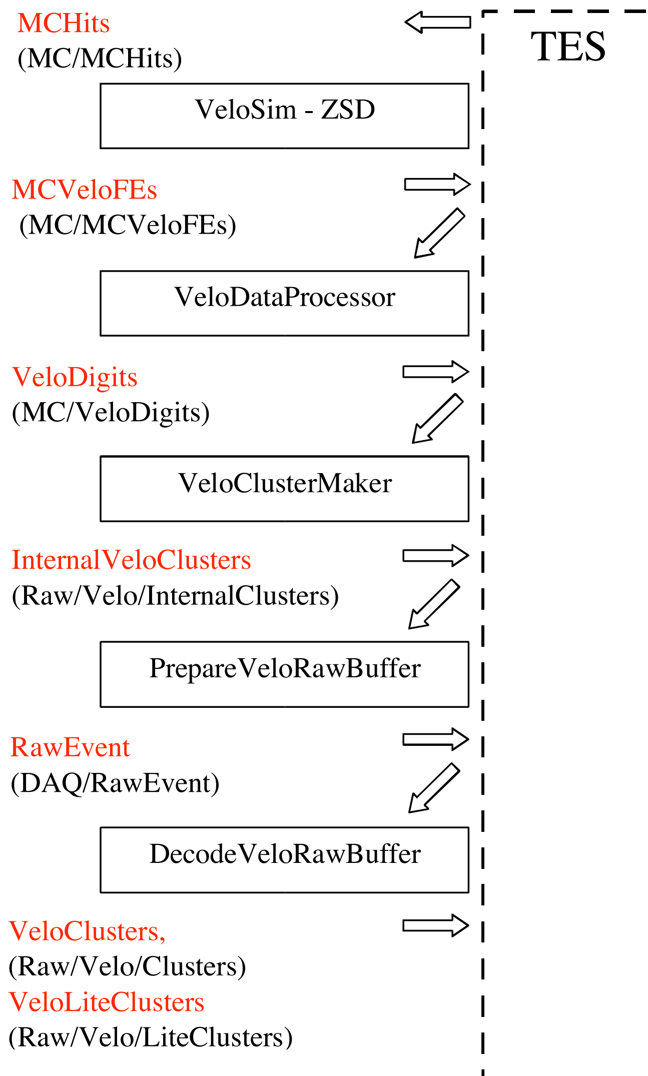

Figure 19 The data flow for the Velo Event Model for DC06.

```
// define interface, header file
typedef LinkerTool<LHCb::VeloDigit, LHCb::MCHit> ascTool;
typedef asctTool::DirectType Table;
typedef Table::Range Range;
typedef Table::iterator iterator;


//retieve the tool, implementation file
// get the digits container
LHCb::VeloDigits* m_digits=
    get<LHCb::VeloDigits>(LHCb::VeloDigitLocation::Default);

LHCb::VeloDigits::iterator digIt;
for(digIt=m_digits->begin(); digIt!=m_digits->end(); ++digIt){
 // retieve the tool, implementation file
 asctTool associator(evtSvc(),
    LHCb::VeloDigitLocation::Default+"2MCHits");
 const Table* table=associator.direct();
 if(!table){
   error()<< " Empty table with associations" <<endmsg;
```

```
    return ( StatusCode::FAILURE );
  }
  Range ran1=table->relations(*digIt);
  iterator it;
  for(it=ran1.begin(); it!=ran1.end(); ++it){
    const LHCb::MCHit* aHit=it->to();
   // do something ...
  }
}
```

5.4 Monitors

As a means for debugging and checking the appropriate monitoring algorithms for each data class have been implemented. Each such algorithm inherits from GaudiTupleAlg and produces a set of control histograms. The monitors are meant to be as simple as possible and basically use only informations available directly from monitored class. Useful example routines are also provided (e.g. how to use the association table with links between VeloDigits and MCHits in VeloDigiMoni).

6 Packaging

The full processing from the simulated MCHits objects to the VeloCluster's ones consists of many phases (silicon simulation, digitization etc), it is thus convenient to split up the algorithms and data classes into separate packages. Such packaging helps not only to understand the data flow but also improves significantly the maintainability of the code. The description of the packages is presented below:

- VeloDigit, VeloLiteCluster and VeloCluster classes are placed in Event/DigiEvent package which is shared between different subdetectors
- InternalVeloCluster is allocated in Velo specific package Event/VeloEvent
- Simulation code, VeloSim, and the appropriate monitoring algorithm (which allows studies of the MCVeloFE object) VeloSimMoni are placed inside Velo/VeloSimulation package
- The remaining algorithms, VeloDataProcessor and VeloClusterMaker, along with the VeloDigiMoni and the VeloRawClusterMoni are inside the Velo/VeloAlgorithms package.
- All routines meant for preparing and decoding of the raw buffer are placed in Velo/VeloDAQ
- Associator algorithms can be found in Velo/VeloAssociators
- Algorithms and data object for decoding of the real raw buffer can be found in Velo/VeloTELL1Alg

7 Conclusion

The Event Model for the Velo detector has been presented. This model provides convenient classes for storing the real and simulated data from the detector.

8 References

[1] G. Haefeli at al, 'TELL1 - specification for a common read out board for LHCb', LHCb-2003-007

[2] B. Jost, N. Neufeld, 'Raw-data transport format', LHCb technical note EDMS 499933

[3] Definition of the RawEvent and RawBuffer classes
http://isscvs.cern.ch/cgi-bin/cvsweb.cgi/Event/DAQEvent/?cvsroot=lhcb

[4] D. Eckstein, 'Velo raw data format and strip numbering', EDMS 637676

[5] G. Haefeli, A. Gong, 'Velo and ST non-zero suppressed bank data format', EDMS 692431

[6] G. Haefeli, A. Gong, 'Velo and ST error bank data format', EDMS 694818

[7] G. Haefeli, A. Gong, 'Velo and ST pedestal bank', EDMS 695007

[8] S. Roiser, 'Gaudi Object Description v2r0 - Documentation'

[9] O. Callot, 'Updated usage of the linker classes', LHCb-2006-008
O. Callot, 'Usage of the linker classes', LHCb-2005-018

[10] R. Jacobsson, 'ODIN raw data format', EDMS 704084