A brief intro to the ATLAS EDM

And some thoughts about LUXE's

Federico Meloni (DESY)

LUXE simulation and analysis TF
13/04/2021

HELMHOLTZ RESEARCH FOR GRAND CHALLENGES

The Event Data Model

A collection of classes, interfaces and concrete types, and their relationship that, together, provide a representation of an event and eases its manipulation by the reconstruction and analysis developers.

The Event Data Model (EDM):

- creates a **commonality** across the detector **subsystems**.
- allows the use of **common software** between online and offline environments.
- defines the **structure of the data at various stages** and allows elements of the data flow processing tasks to access the data without resorting to the use of other resources, e.g. databases.
- defines **additional metadata** that is added to the detector data allowing processing tasks to quickly identify the type and origin of each event.

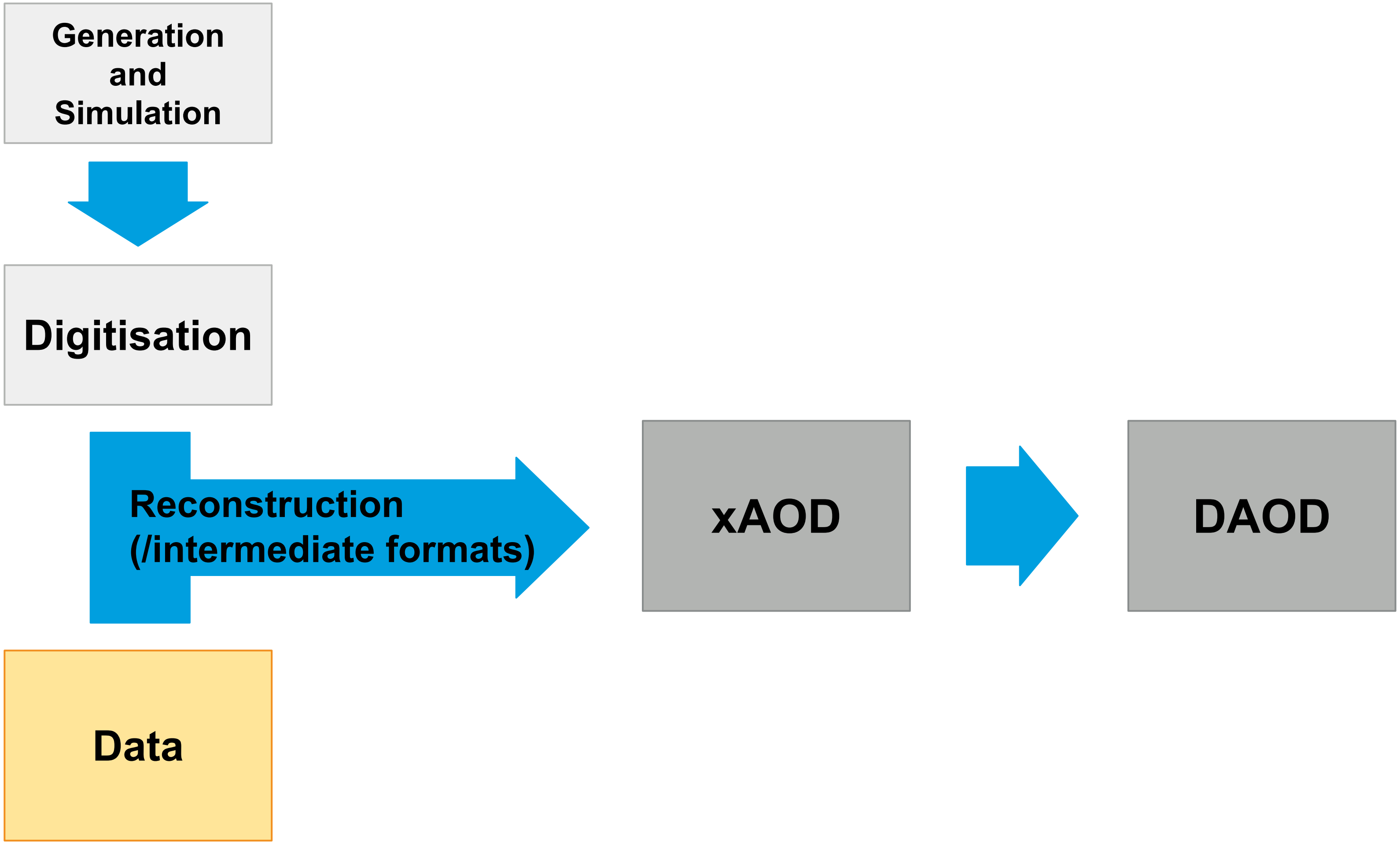
The ATLAS EDM

The ATLAS detector produces ~ one Petabyte of data per year, a vast amount of information which prohibits the wide distribution of raw data to worldwide collaborators.

To analyse the data two additional stages of datasets are introduced:

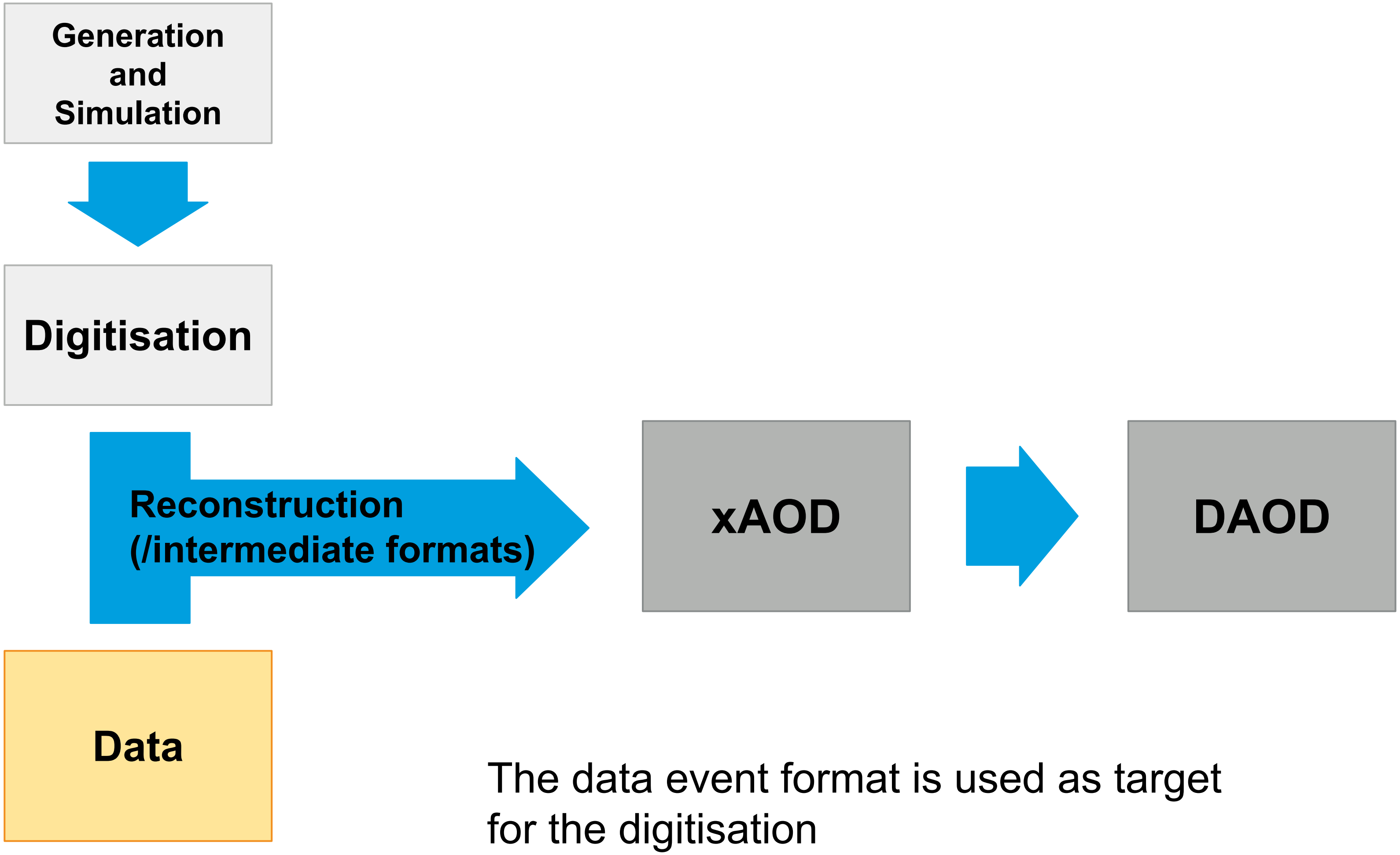
- The Analysis Object Data (xAOD) which is a summary of the reconstructed event, and contains sufficient information for common analyses. This format is readable by both the ATLAS experiment software framework (Athena) and ROOT.
- The AOD derivations (DAOD) these are xAODs that had some of the information filtered away (skimming, slimming, thinning...) and other information added (reconstructed jet collections, flavour tagging information, ...)

(Simplified) ATLAS data processing



Generation and Simulation
Digitisation
Reconstruction (/intermediate formats)
Data
xAOD
DAOD

(Simplified) ATLAS data processing


Generation and Simulation
Digitisation
Reconstruction (/intermediate formats)
xAOD
DAOD
Data

The data event format is used as target for the digitisation

Raw data format requirements

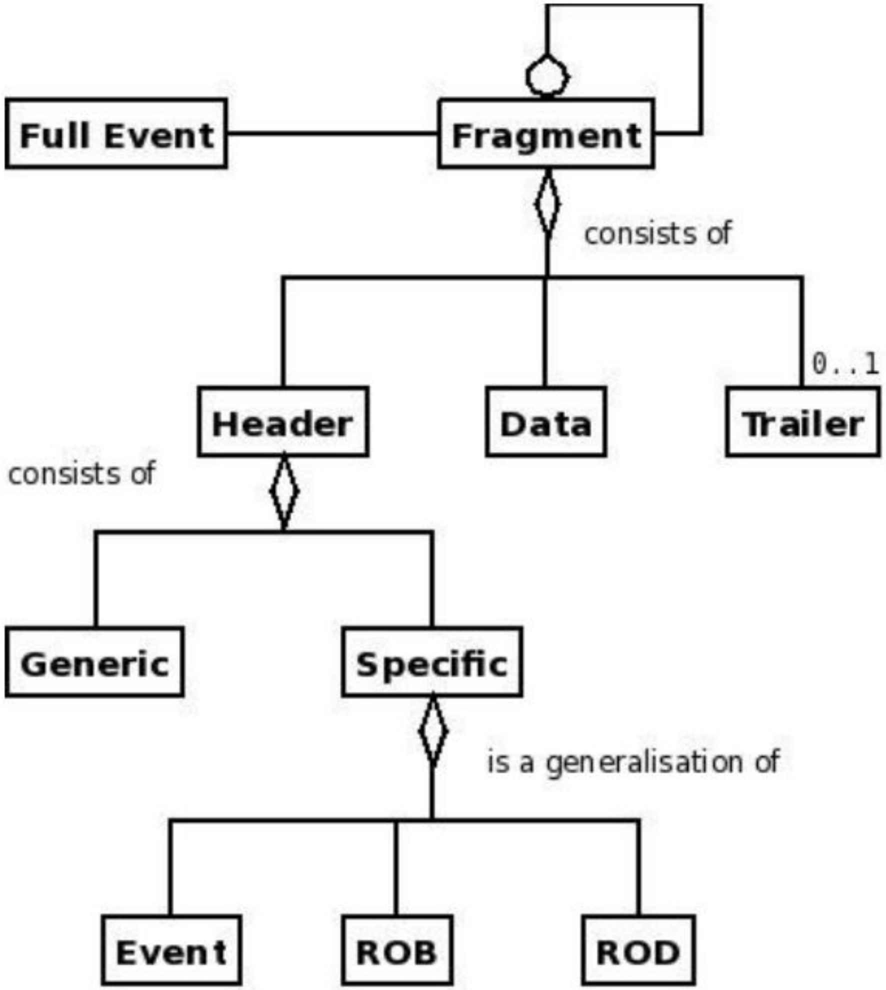
The event format shall fulfil the following requirements:

- The **size** of an event should be **free to increase or decrease** depending on the specific data taking configuration;
- There shall be **no minimum or maximum event data size** implied by the format;
- The event format should provide information redundancy to **allow self consistency checks** of the event to be made;
- The event **formatting** information shall **not exceed 20%** of the typical full ATLAS event data size;
- The event format should be **modular**;
- The **basic unit should be a fragment**. Fragments are: parts of an event coming from a part of the readout chain or the (Full) Event itself;

Raw data format requirements

- The fragments should have **identical structure**;
- The event format shall facilitate the identification of fragments;
- The event format shall provide an **event header**;
- The event format shall provide the event identifier and trigger type within the full event header;
- The event format shall provide a means of identifying whether the event has been corrupted during transmission within the Data Flow, e.g. DMA time-out, truncation etc;
- The event format shall provide a means of identifying whether the event has been corrupted due to hardware problems, e.g. a bit error.

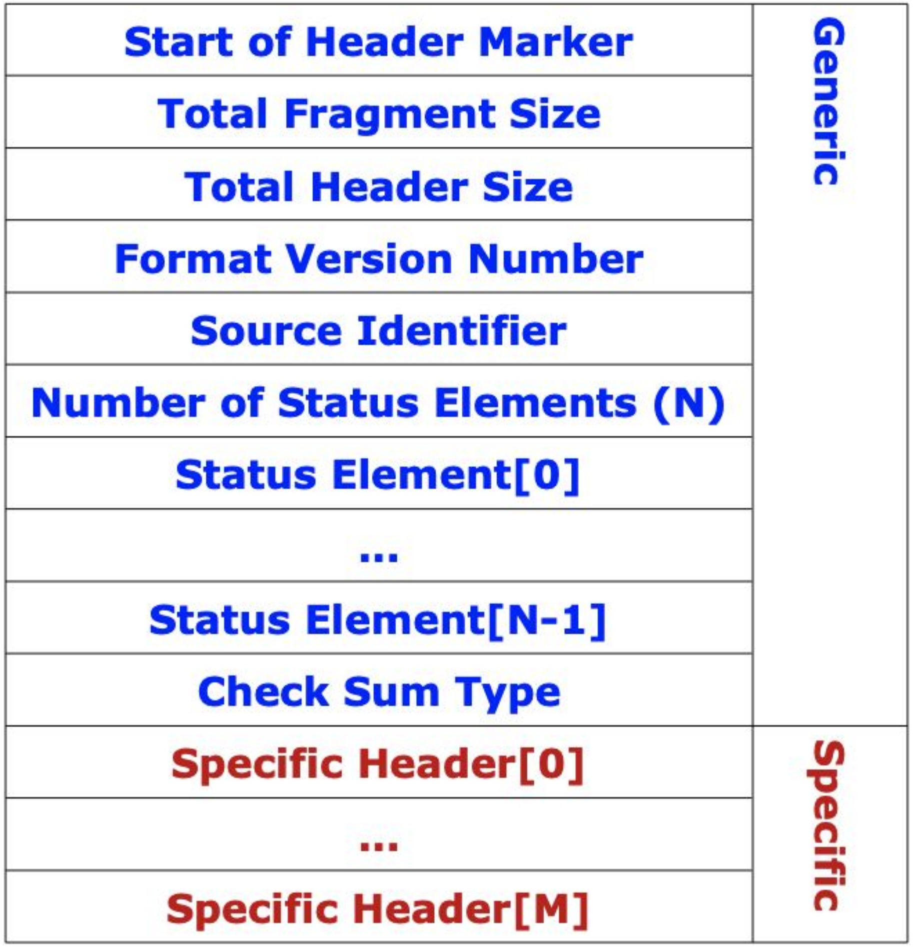
RAW Event structure



A full event, divided in fragments respecting the requirements.

Data headers

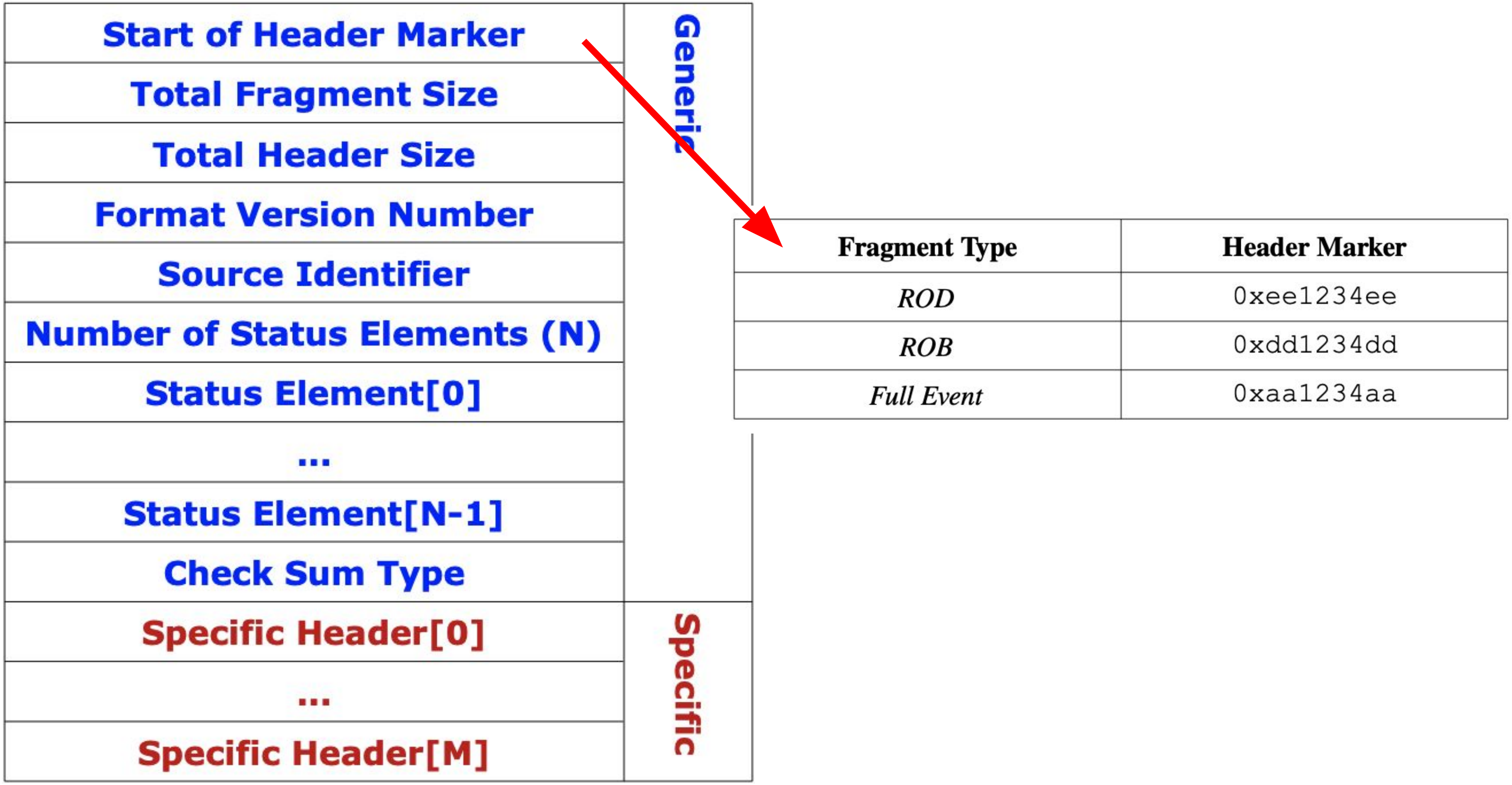
Quickly zooming on the per-fragment metadata



Start of Header Marker	Generic
Total Fragment Size	
Total Header Size	
Format Version Number	
Source Identifier	
Number of Status Elements (N)	
Status Element[0]	
...	
Status Element[N-1]	
Check Sum Type	
Specific Header[0]	Specific
...	
Specific Header[M]	

Data headers

Quickly zooming on the per-fragment metadata

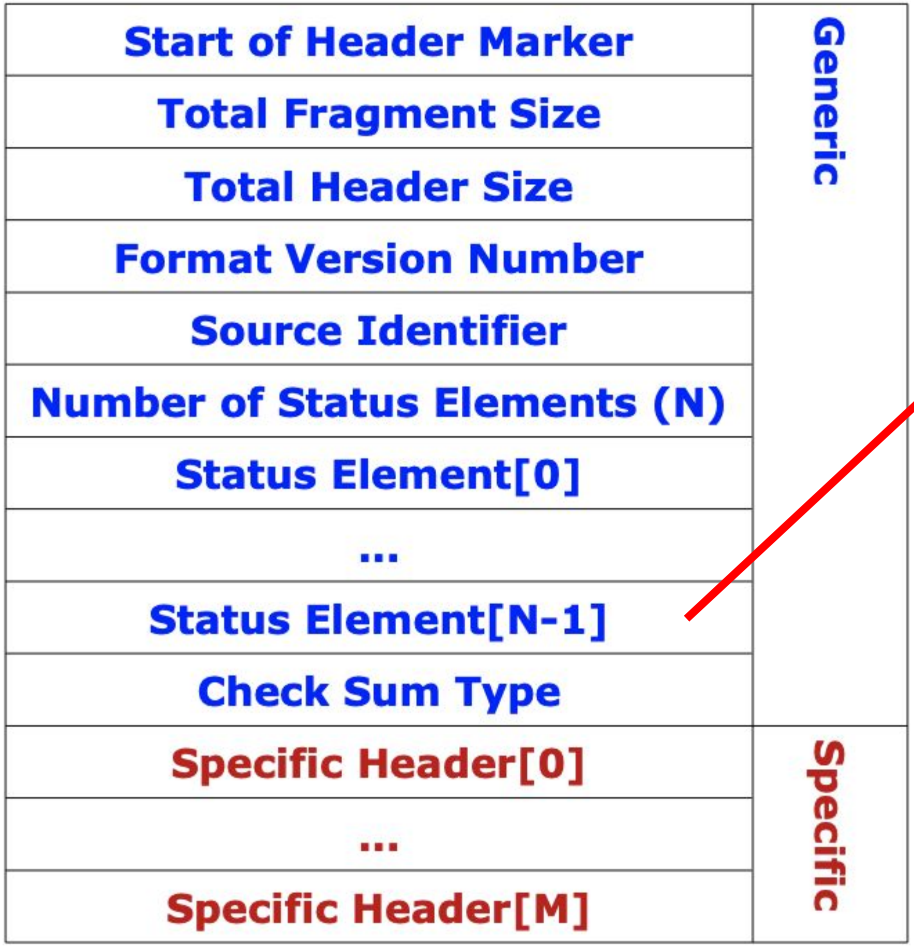


Start of Header Marker	Generic
Total Fragment Size	
Total Header Size	
Format Version Number	
Source Identifier	
Number of Status Elements (N)	
Status Element[0]	
...	
Status Element[N-1]	
Check Sum Type	
Specific Header[0]	Specific
...	
Specific Header[M]	

Fragment Type	**Header Marker**
ROD	`0xee1234ee`
ROB	`0xdd1234dd`
Full Event	`0xaa1234aa`

Data headers

Quickly zooming on the per-fragment metadata



Start of Header Marker

Total Fragment Size

Total Header Size

Format Version Number

Source Identifier

Number of Status Elements (N)

Status Element[0]

...

Status Element[N-1]

Check Sum Type

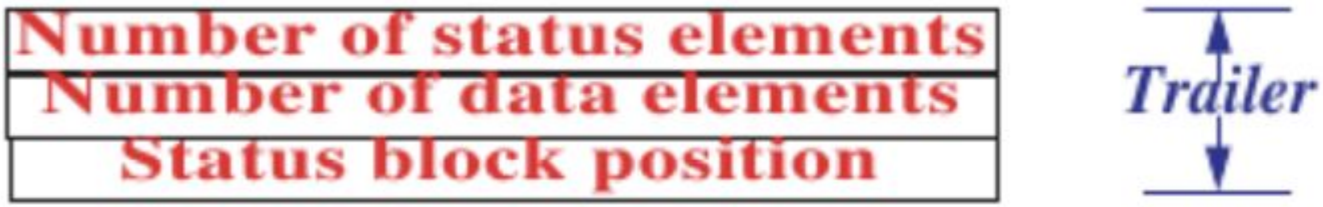
Specific Header[0]

...

Specific Header[M]

Generic Field Value	*Description*
0 (0x00)	Unclassified
1 (0x01)	An internal check of the BCID has failed.
2 (0x02)	An internal check of the EL1ID has failed.
4 (0x04)	A time out in one of the modules has occurred. The fragment may be incomplete.
8 (0x08)	Data may be incorrect. Further explanation in the Specific field.
16 (0x10)	An overflow in one of the internal buffers has occurred. The fragment may be incomplete.
32 (0x20)	SW dummy fragment: the fragment was generated in a SW component.
64 (0x40)	Reserved
128 (0x80)	Reserved

Data trailers



Number of status elements

Number of data elements

Status block position

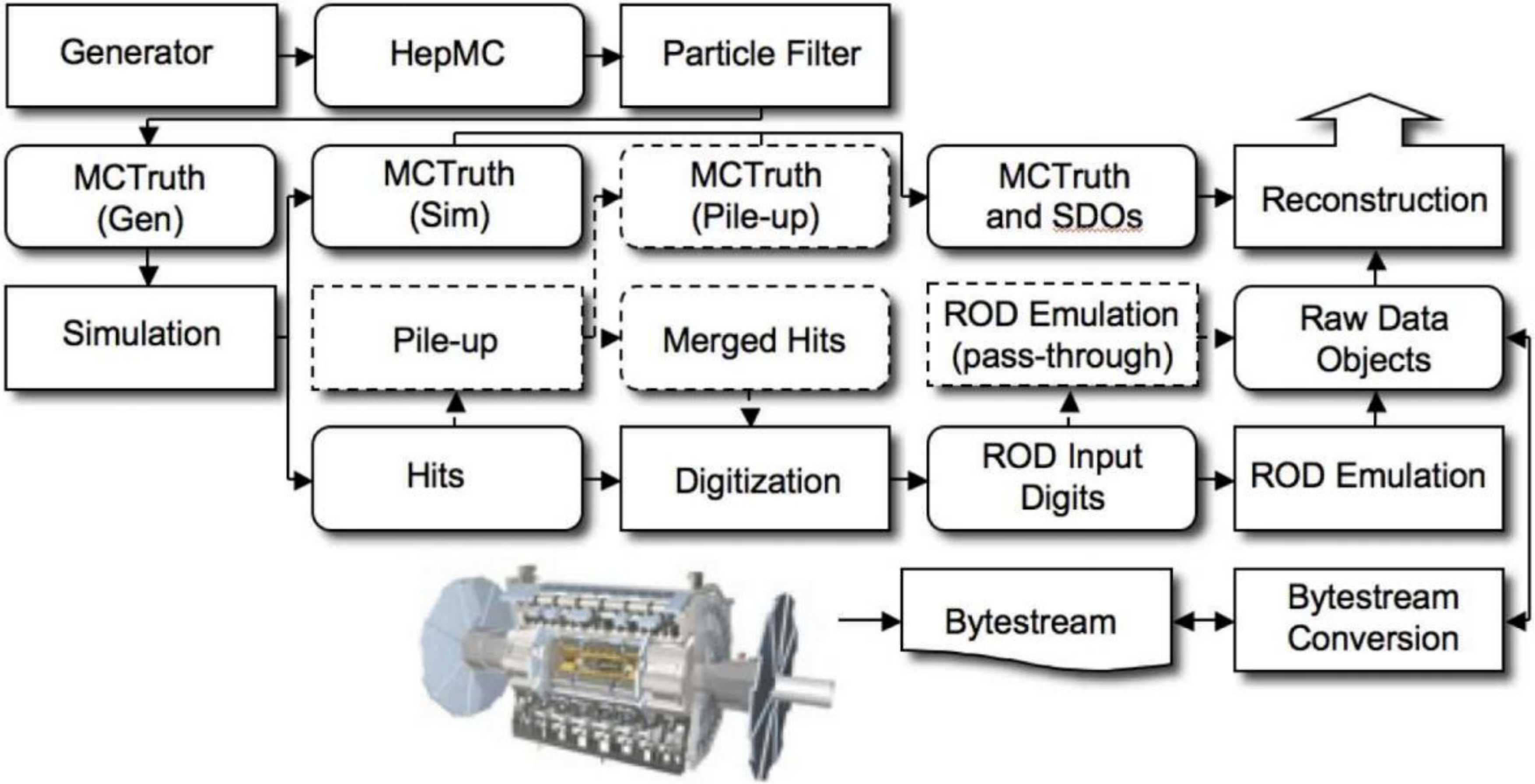
The trailer contains:

- Number of data elements
- Number of Status elements
- status block position (some detectors prefer to ship the status in front of the data)

arXiv: 1005.4568

What about simulation?

The path to replicating the RAW data format



Simulation and digitisation

The output from the simulation is a **hit file**, containing some **metadata describing the configuration** of the simulation during the run, all requested truth information, and a **collection of hits for each subdetector**.

- The hits are records of energy deposition, with position and time, during the simulation.
- Each subdetector implements their own sensitive detector for the selection, processing, and recording of these hits.
  - In the calorimetry, there are far too many hits for the individual storage of a four-vector for each. Hit merging occurs at the end of each event.

The ATLAS digitization software converts the hits produced by the core simulation into detector responses: “digits”.

The **various subdetector digitisation packages** are steered by a top-level Python digitization package that ensures uniform and consistent configuration.

MC truth information

The Geant4 simulation adds to the Monte Carlo truth record already defined during generation.

- Too many secondary tracks are produced during detector simulation to store information for every interaction.

Only those interactions which are of greatest relevance to physics analyses are saved.

- Most are applicable only to the inner detector.
- For each interaction that satisfies any of the storage criteria, the incoming particle, step information, vertex, and outgoing particles are included in the truth record.

Later in the software chain, individual track segments are recombined.

Simulation metadata

Two databases are used to construct the detector geometry (based on GeoModel, then translated to the Geant4 format at runtime):

- The ATLAS Geometry database - to store basic constants (dimensions, positions, rotations, magnetic field).
- The ATLAS Conditions database - to store various conditions data (e.g. calibrations, dead channel, misalignments).

At CERN, Oracle databases are used. With any stable software release, a subset of data needed for Athena jobs is replicated from Oracle into SQLite and is distributed to the production centers.

- These files can also be replicated to individual production nodes for local and rapid access.
- The database replica version to be used can be chosen at run time.

Event Reconstruction

The Raw Data Objects from either data or simulations are processed by a common reconstruction software.

ATLAS has two types of sub-detector systems:

- **trackers** (the Inner Detector and Muon Spectrometer), which measure momenta of charged particles;
- **calorimeters** (Tile and Liquid Argon), which measure energy depositions.

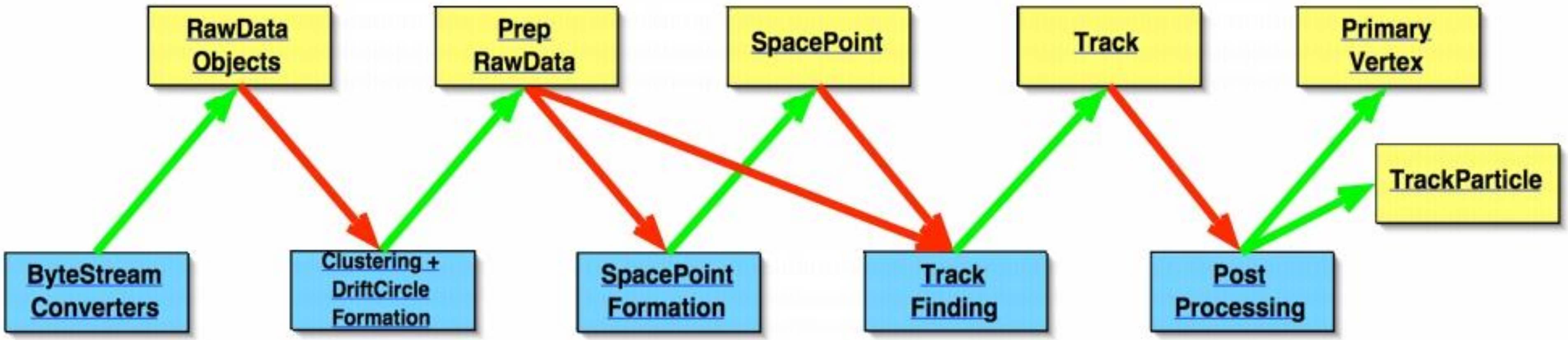
As mentioned before, a major aim in the design of the EDM is to share as much code as is possible within these common sub-system types.

Trackers

The EDM supports different tracking devices with shared code.

A common track class and a standard definition of:

- Track parameters (on all the various surfaces found along the track);
- Interfaces to hit-clusters, drift circles, etc;



Tracking must handle many **different coordinate frames**, as a track can have measurements on many different surfaces.

- Some sub-detectors return one-dimensional measurements, so these must be combined to form two-dimensional SpacePoints.

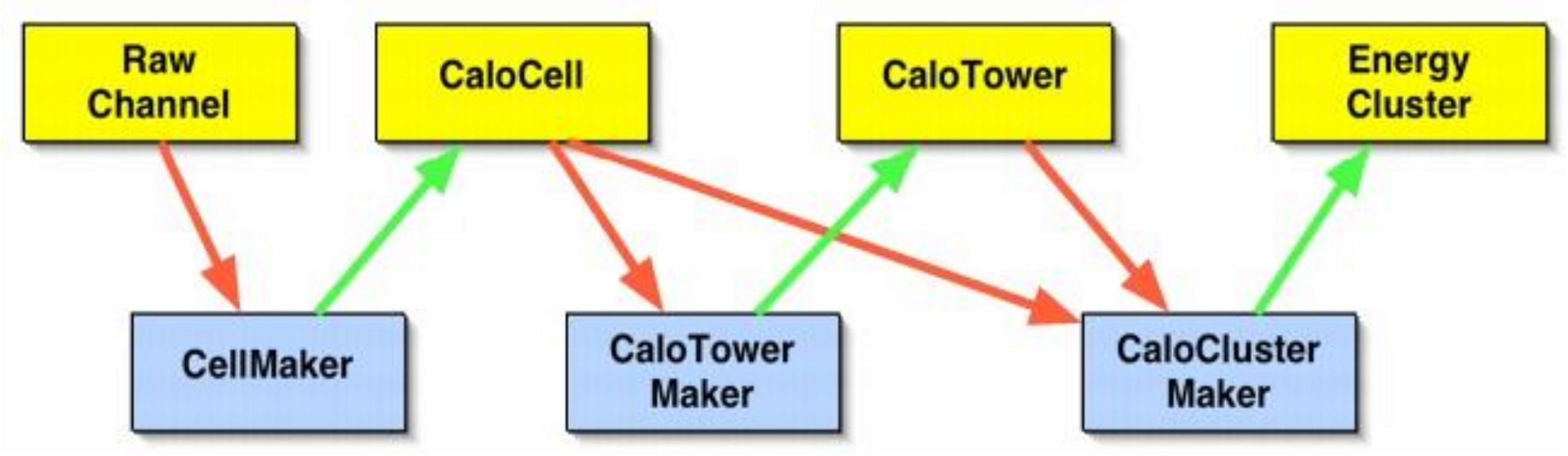
Tracks can then be reconstructed and used to find vertices, or as inputs to

Calorimeters

The two types of calorimeter have different data formats at the raw data level.

- The EDM uses one common calibrated input object, CaloCell.
- Neighbouring CaloCells are used to produce higher level objects, such as "clusters" (collections of calorimeter elements).
- A navigation scheme allows access to constituent data objects (CaloCells).

All calorimeter data classes inherits from a four-momentum interface which allows the use of common tools only requiring kinematic information.



The xAOD

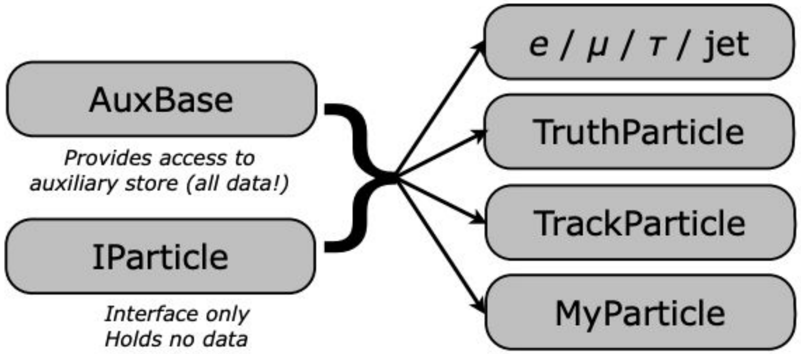
The xAOD represents the “analysis-level” EDM of the ATLAS experiment:

- Usable in both full **Athena** jobs, **and** in very lightweight **ROOT** jobs.
- The previous AOD EDM could essentially only be used in Athena, which limited its analysis use considerably.
- The EDM is mainly **designed for analysis use**.

Be as fast as possible:

- Simple ROOT ntuples will always be the fastest, but it was designed to be as close to them as possible.
- The user is allowed to add/remove variables to/from objects (containers) at runtime

xAOD characteristics


AuxBase
Provides access to auxiliary store (all data!)
IParticle
Interface only Holds no data
e / μ / τ / jet
TruthParticle
TrackParticle
MyParticle

Objects are split in two:

- **Interface objects**: these are objects that provide a usable UI for the type, but don't hold on to any persistent data.
- **Payload objects**: these objects (called "Auxiliary stores") that just allocate contiguous memory for the data, and allow the interface objects to access this data.

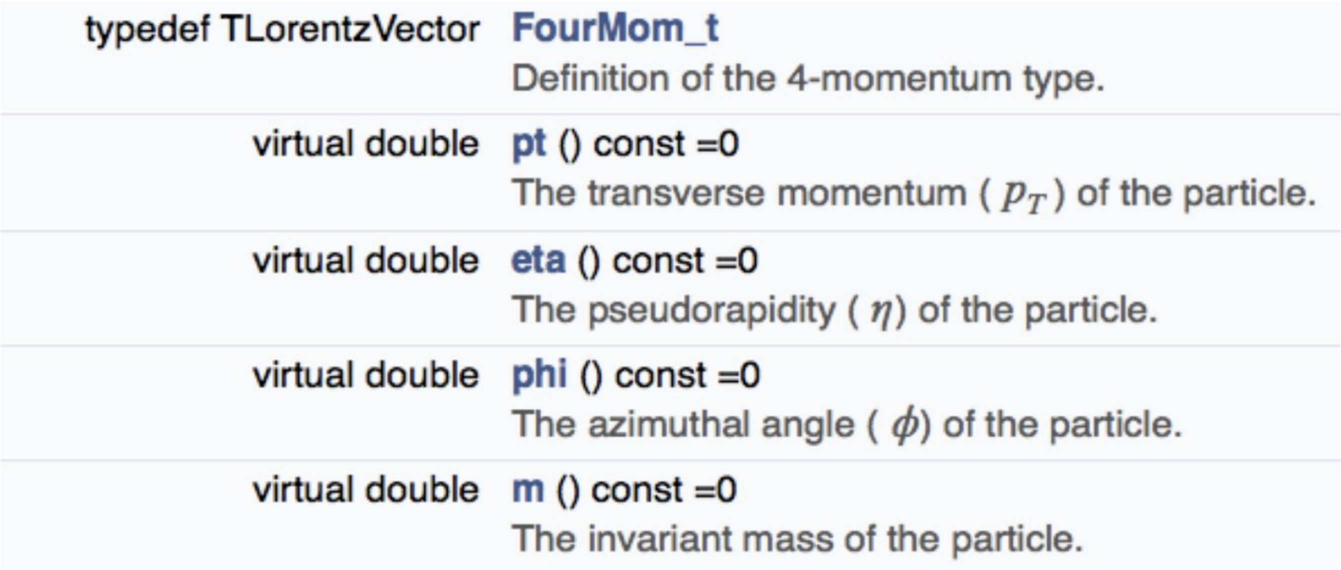
This split means that:

- Data **storage technology is independent of the interface** that we provide for a specific class (i.e. muons).
- We can **partially read** information about objects during analysis.
- The user code should only create payload objects (when creating new xAOD objects), but never manipulate them directly.

An important xAOD class: xAOD::IParticle

This class is a pure virtual interface for:

- All particle types (electron, muon, jet, …)
- Clustered energy deposits in the calorimeter
- Charged particle trajectories (tracks)
- Provides access to the basic particle properties, like their four momentum

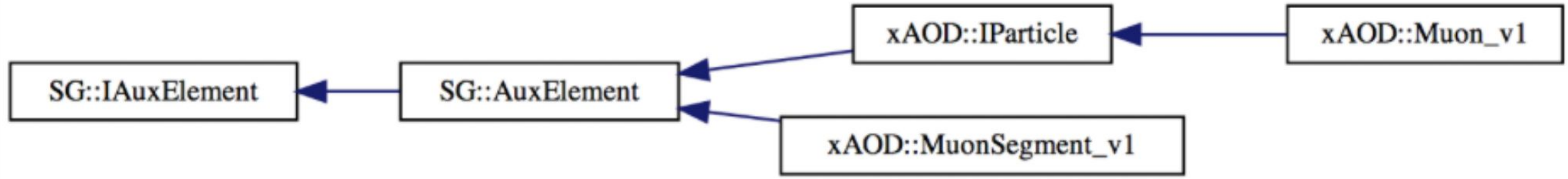


typedef TLorentzVector	FourMom_t Definition of the 4-momentum type.
virtual double	pt () const =0 The transverse momentum (p_T) of the particle.
virtual double	eta () const =0 The pseudorapidity (η) of the particle.
virtual double	phi () const =0 The azimuthal angle (ϕ) of the particle.
virtual double	m () const =0 The invariant mass of the particle.

xAOD class inheritance

xAOD::IParticle is not at the base of the inheritance structure.

- The class that all xAOD interface classes inherit from is SG::AuxElement



It's this class that provides the core infrastructure for the xAOD EDM implementation.

- Provides **standardised access** to all variables stored in the object's auxiliary store
- Allows the user to add **new variables to an object at runtime**

Storage and linking

Almost all EDM objects are **stored in containers**.

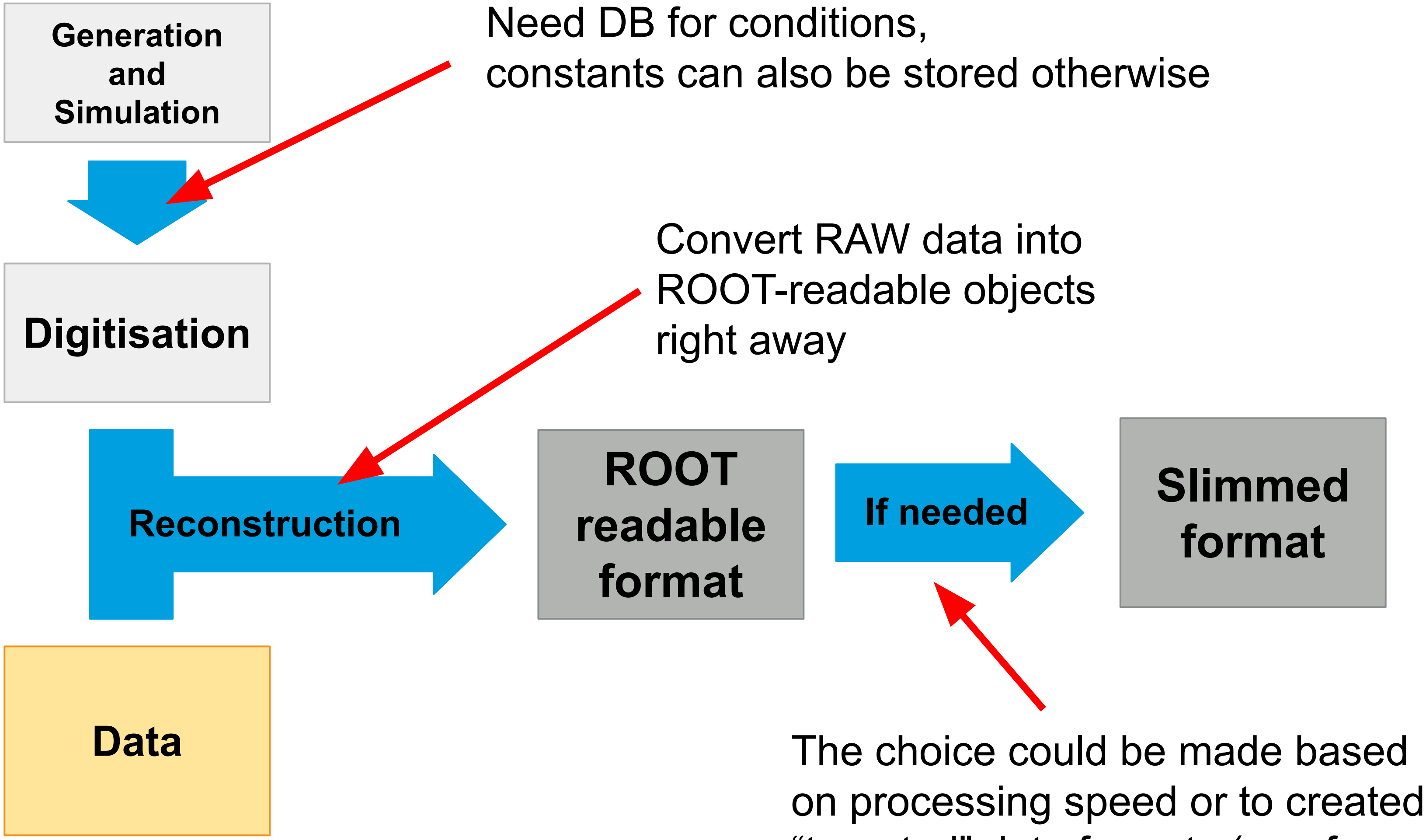
The containers use a custom vector type, DataVector<T>.

- For all simple intents and purposes DataVector<T> behaves just like std::vector<T>.
- Needed so we can use **polymorphism inside the containers**.
- Includes code for implementing the interface <-> auxiliary store separation and allows us to define inheritance relationships between the types.

Objects can point to each other.

- **Save space** in memory and on disk.
- High level objects point back to their constituents (calorimeter clusters and track particles).
- To write these links to disk we use the **ElementLink<T> smart pointer** type (ElementLink is two numbers: one identifying the container of the target object and the second one storing the index of the target in its container)

What about LUXE? (1/3)



Need DB for conditions, constants can also be stored otherwise

Convert RAW data into ROOT-readable objects right away

The choice could be made based on processing speed or to created “targeted” data formats (e.g. for the BSM detector)

What about LUXE? (2/3)

Adopt many ideas also present in ATLAS' xAODs.

- ROOT readable, with two possible options:
  - Loading of EDM classes for advanced use (links, etc)
  - Fully flat ROOT-only is in principle also possible but quickly becomes very messy (interfaces, functions, ...)

- Implementation of classes as interface + auxiliary store
  - Keep access to variables standardised
  - Avoid complex object oriented EDM classes
  - Simple class inheritance structure

- Structure information in containers
  - New containers can be added at runtime.
  - New variables can be added to an object at runtime.

What about LUXE? (3/3)

- The reconstruction steps will be equivalent to adding containers and links (no change in file format)

- The EDM should allow for the analysis to be performed with both C++ and/or python (though columnar analysis would not be trivial with containers).

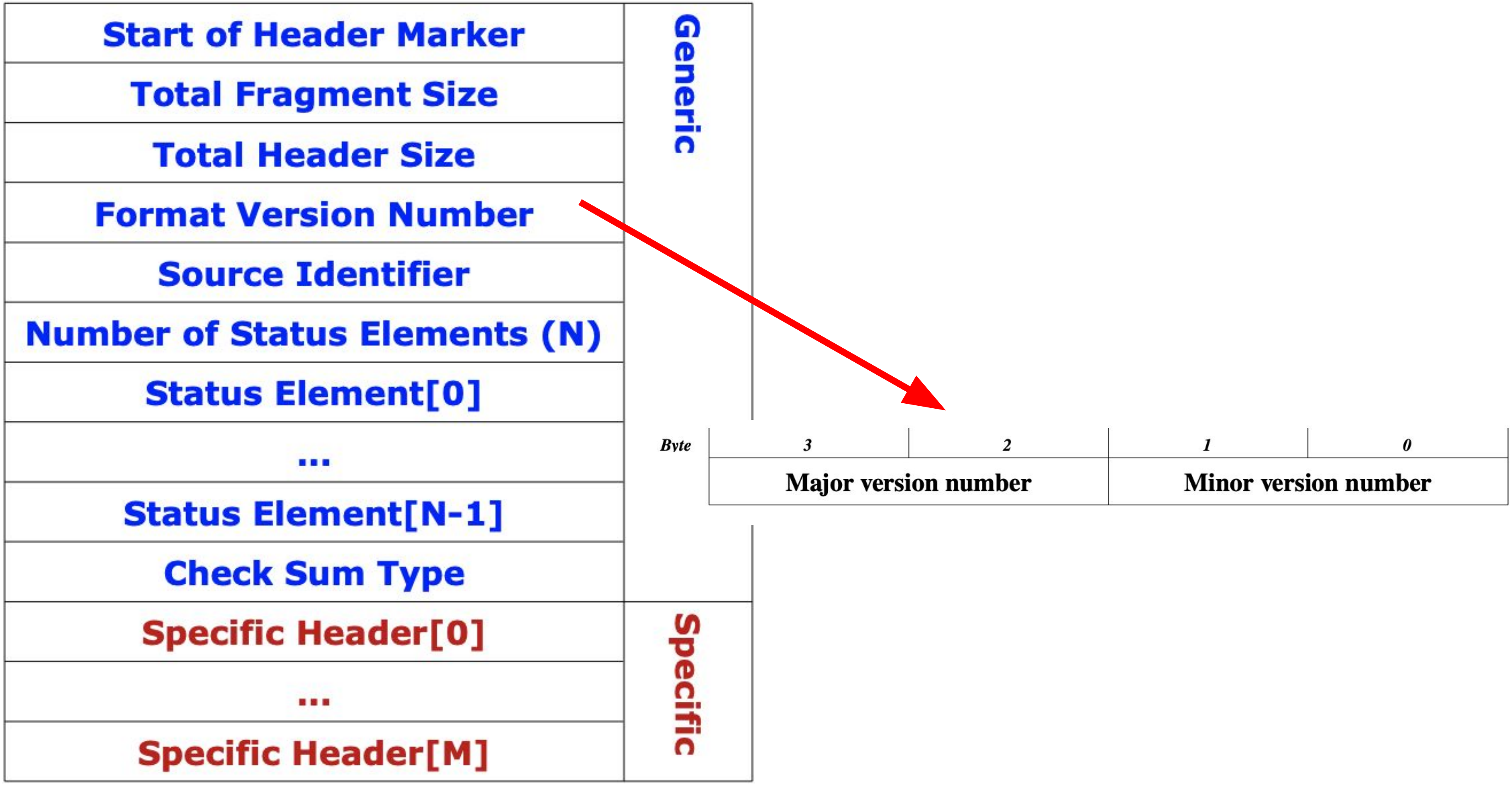
- Content can be inspected in ROOT::TBrowser.

- Warning: ROOT-readability means that files written with a certain version of the EDM may not be readable with more recent versions of the software (if changes are made to the class templates).

Extras

Data headers

Quickly zooming on the per-fragment metadata



Start of Header Marker

Total Fragment Size

Total Header Size

Format Version Number

Source Identifier

Number of Status Elements (N)

Status Element[0]

...

Status Element[N-1]

Check Sum Type

Specific Header[0]

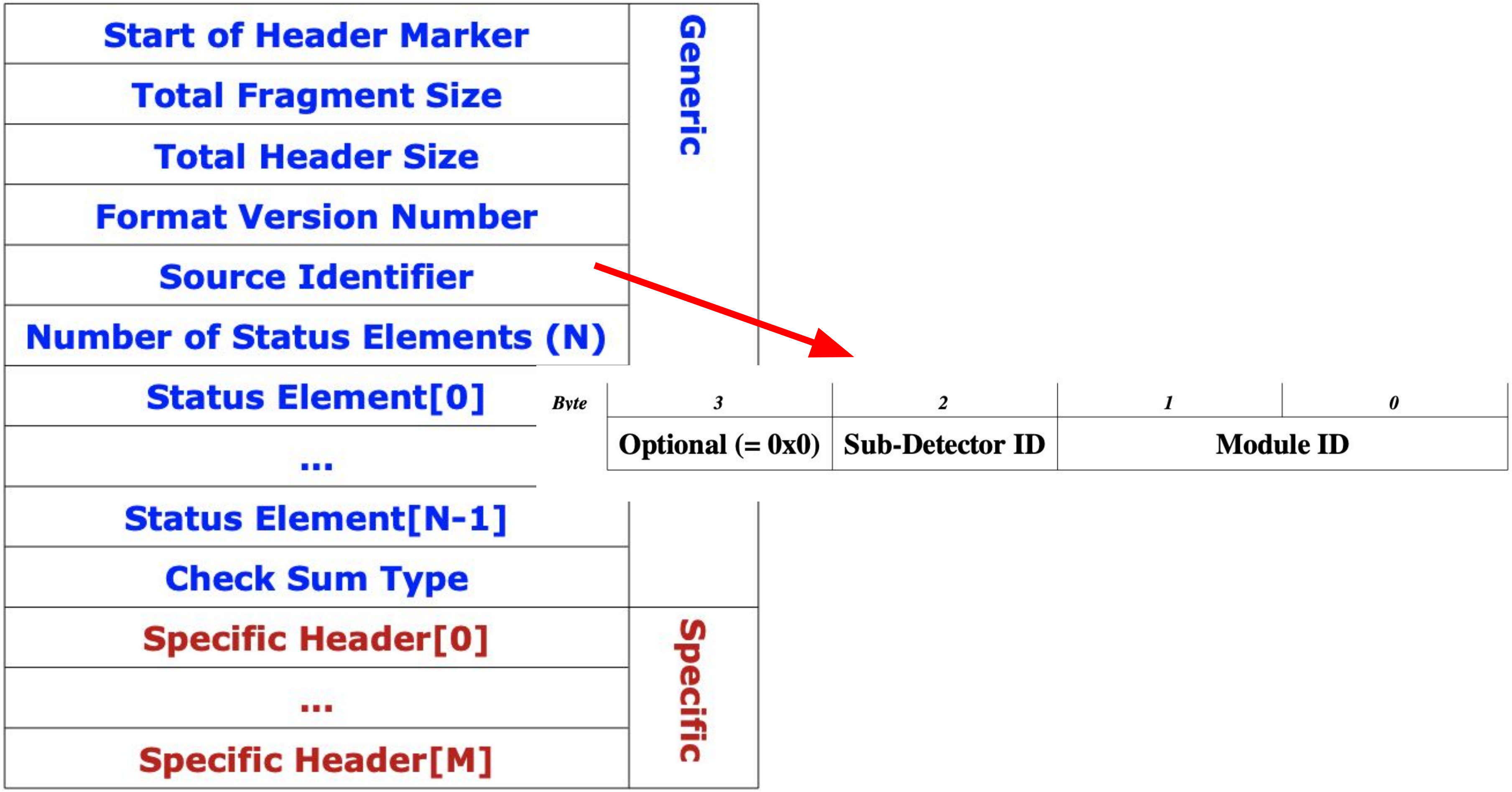
...

Specific Header[M]

Byte	*3*	*2*	*1*	*0*
	Major version number		Minor version number	

Data headers

Quickly zooming on the per-fragment metadata



Start of Header Marker	Generic
Total Fragment Size	
Total Header Size	
Format Version Number	
Source Identifier	
Number of Status Elements (N)	
Status Element[0]	
...	
Status Element[N-1]	
Check Sum Type	
Specific Header[0]	Specific
...	
Specific Header[M]	

Byte	*3*	*2*	*1*	*0*
	Optional (= 0x0)	Sub-Detector ID	Module ID	