Software week, 10-12 November 2004

Olivier Callot

Event Model and Relations

- **What is the event model**
  - Constraints and requirements, persistency
  - Data access, containers
- **XML, GOD and dictionaries**
  - Building new classes
- **Relations**
  - With Linker

Event Model

What is an Event Model ?

- **Standard description of data classes**
  - C++ objects containing data
    - Basically only setters and getters
- **Allows persistency**
  - Objects can be written on storage, and read back by another program
  - The way to convert them must be specified.
    - Streamers = piece of code to put the bytes of the objects one after the other (writing) or to reconstruct the object from a byte stream.
      - ➔Old implementation with ROOT, abandoned in 2003
    - Automatic conversion with dictionaries
      - ➔Standard POOL technology, implies that we have a dictionary for each object and collections of it.
  - Imposes some constraints of what can be in an object

- **Reference between objects are special**
  - **A simple C++ pointer can be written with POOL**
      - ➔This is new compared to old Gaudi persistency
    - Works only if the pointed object is owned, and known only by the parent object.
  - Gaudi has **`SmartRef`** objects to handle references between autonomous objects
    - It behaves like a pointer for normal usage.
    - But it makes a logical reference to the pointed object when written
    - This reference can be transformed back to a pointer when reading data.
      - ➔In fact identifies an object by the **`container`** it belongs to, plus its position / key in the container
    - Example: **`MCVertex`** has a list of **`MCParticles`**, and **`MCParticle`** has a parent **`MCVertex`**…

- **Hand written creation of objects is discouraged**
  - We have a tool to generate the C++ header and the dictionary, starting from a formal description in XML: **GOD.** This is an acronym for Gaudi Object Description, and will be discussed later in this presentation
  - This tool gives a standard layout of setters and getters, handles relations and builds all the ingredients needed for persistency.

How do we use it ?

- **Event classes are in dedicated packages**
  - Usually under the `Event` hat, e.g. `Event/CaloEvent`.
  - These are shared by most applications
    - They are in the high level “project” LHCb, used by the various applications Boole, Brunel, DaVinci, Panoramix.
- **These packages must be stable**
  - Change in the object implies difficulties to read back data produced with previous versions
    - In principle, POOL provides a way to handle this “schema evolution”, but with strong restrictions, in practice only addition of new information will most probably work, but modification may work only in very limited cases
  - Objects must then be **WELL DESIGNED**.
    - Reviews and discussions with experts are **mandatory**.

- **Event classes must be identified by a classID**
  - This is how the system knows the type of the object
  - These ID are allocated by Marco, see the Event Model web pages for a list of the known ones.
  - Using a wrong ID can be annoying…
- **But "private" event classes are possible also**
  - If they are never written, the constraints are smaller
    - Schema evolution is limited to a complete recompilation of the relevant application
    - No need for a real ID
  - But frequently this is to build a prototype for a future persistent object.
    - Follow the rules, and your object can be upgraded to an 'official' object whenever needed.

Containers

- **Objects can exist as a single instance**
  - Single copy per event, e.g. EventHeader
  - It is then accessed on its own
    - The location to access it is specified in the event model
    - Usually this is something like `EventHeaderLocation::Default`
    - Other locations can be pre-defined, changing the name after ::
- **A more frequent use case is a collection of objects**
  - Digits, clusters, tracks, particles,…
  - Objects are placed in a **container**, and can be accessed only via the container
    - In fact the object is created by `new` and the pointer to the object is in the container
    - This is just technical: The object belongs to the container. The container is the only access path to the object.

Keys

◆It is very useful to have an ID for each object

- We call it a **key**, as this is one way to access a specific object.
- Objects of this type are in a special container of type **`KeyedContainer`**.
  - This gives some extra functionality, like ‘give me the object with that key’, with the method **`container -> object( key);`**
- Of course **the key must be unique**…
  - This is guaranteed: When inserting an object into the container, the uniqueness is checked, and an exception thrown if needed.
- It can be **automatically assigned**
  - When creating a track or a particle, the key is assigned as a serial number in the container, starting at zero. The key has no special meaning.
- It can be specified
  - A VeloCluster has its first VeloChannelID as key.

Access to data objects

◆Access is by the location on TES

- TES means “**T**ransient **E**vent **S**tore”
- This is a piece of memory managed by the **`EventDataService`**, where objects are identified by their ‘location’
  - Resemble a file system, with a tree structure.
- To get an object, one has to **specify its type and its location**
  - To retrieve either a simple data object, or a container

```
TrgTracks* myTr = get<TrgTracks>( TrgTrackLocation::Velo );
```

➔The argument is a string,

➔A good practice is to have a member variable there, initialized in the creator to a proper default, and with a property to change it by job option.

- **`exist`** check if the object exists, sometimes useful as **`get`** throw an exception in case it doesn’t exist…

◆Missing or Empty containers ?

- We make a strong difference between an empty container and a missing container
  - Empty = the algorithm in charge of creating the objects has run, but hasn't produced any object
  - Missing = the algorithm in charge has not run / has failed
    - ➔This is an abnormal condition, and the job should abort.

◆More complex access

- Gaudi has more basic ways, described in old documentation.
  - They are discouraged, you should use the '**`get`**' and '**`exist`**' methods of GaudiAlgorithm / GaudiTool from now on.
- If you see them in existing code, please report to the author of the code: Conversion to **`GaudiAlgorithm`** or **`GaudiTool`** was probably not yet done on this code !

◆Storing an object is even simpler

```
put( object, location);
```

- **`object`** is a pointer to the object, created by **`new`**, which can be a container
- **`location`** is a string, usually **`ObjectLocation::Default`**.
- An exception is thrown if an object with this name already exists.

◆Accessing keyed objects

- First retrieve the container
  - Note: When a keyed object 'Example' is defined, the event model specifies also the type of the container by putting the name plural
    - ➔Example → Examples is a **`typedef`** for **`KeyedContainer<Example>`**
    - ➔Vertex → Vertices
- Then access the object in the container
  - Frequent use: iterate on the all objects in the container
  - But access by key is possible

- To access all objects in a container

```
MyObjects* container = get<MyObjects>( m_location );
for ( MyObjects::const_iterator it = container->begin();
      container->end() != it; ++it ){
  MyObject* obj = *it;
   ... Work with obj ...
}
```

- To access an object knowing its key

```
MyObjects* container = get<MyObjects>( m_location );
MyObject* obj = container->object( aKey );
```

◆Creating and populating a container is simple

```
MyObjects* container = new MyObjects();  // create
put( container, m_location );  // register in TES

for ( ... ) {
  ...
  MyObject* obj = new MyObject();  // create object
  ...
  container->insert( obj );
  container->insert( obj, key );  // to specify the key
}
```

Ownership

This is a delicate question

- In principle, the creator of an object should take care of its deletion, to avoid memory leak
  - Non deleted objects use memory, that can not be re-used !
- EXCEPT that objects on the TES are cleaned up at the end of the event by the store itself
  - Registering an object in the TES implies a transfer of ownership !

NEVER ever even think of …

- Deleting a container in the TES
- Deleting an object in a container
- And even modifying an object in the TES.
  - You are not the owner ! Other algorithms may expect it…

Data On Demand

New feature of Gaudi

- Some data is NOT on the TES, is not on the input file, but we know a method to produce the data if needed
  - Example: re-create clusters from the RawBuffer in DaVinci

Triggered by accessing a non-existent TES object

- If not in the TES, if not in the input file
- Ask the **`DataOnDemandSvc`** if he knows how to produce this named object
- If yes, the relevant algorithm is executed, and the created data is returned to the user
  - One cannot see if the data was read, or produced on the spot
- Of course the pair ‘object name’ + ‘algorithm’ must be specified to the service by job options.

- **Usage has started**
  - In DaVinci
    - For decoding Calo data only when needed.
  - In Brunel
    - For decoding the RawBuffer

- **Will probably (soon) replace the algorithm specification in the Associator package**
  - This was a ‘manual’ implementation of the same functionality

- **May become a default for accessing raw data**
  - Decoding from RawBuffer on demand.
  - Avoids repeating the same raw data in two formats, a compact one (RawBuffer) and a verbose one (container of cluster objects).

XML and GOD

“In GOD we trust”

Stefan Roiser, the father of GOD

If I can say !
Or godfather ?

XML

What is XML ?

- This is a language to describe ‘objects’
- Based on **elements** to identify the entities
- Attributes can be attached to elements
- Elements have a name

Syntax (short version !)

- “**<**“ and “**>**” are used to mark the structured part of the text
- Entities have a name. Two forms exist
  - `<name  attribute = “value”  />`
  - `<name attribute = “value” > …something…  </name>`
  - The first form allows to specify a few properties inside the entity
  - The second form allows defining complex entities, with a lot of structures inside, like other entities and so on.

Inside an entity, one specifies attributes

- Syntax is simple
  - `Key = “value”`
- This means only text values.
- No separation between successive keys

```
name = “test” type = “int”
desc = “This is the test number”
```

- Line breaks are not relevant
  - Some people like to have one attribute per line, vertically aligned
  - The tools to edit xml files (**XmlEditor** or **emacs** menu) tend to put everything on the same line, even what was previously vertically aligned…

Indentation helps to see when an entity ends…

- Automatic inside **emacs**.

Some characters are reserved

- < > & are used for the syntax of xml
- If you want to use then as character, you have to specify them by name:
  - `&gt;` gives the character “>”
  - `&lt;` gives the character “<“
  - `&amp;` gives the character “&”
  - `&quot;` gives the character “
  - `&apos;` gives the character “’”
- This is mainly used when putting C++ code fragments in XML
  - You can imagine that this becomes unreadable quite fast !
    - ➔Try to code

```
if ( a && b ) info() << “ a = “ << a << endreq;
```

- XML compiler diagnostics were poor on that, but this is now fixed in the recent versions.

Comments have beginning and end tags

- `<!--` is the begin tag
- `-->` is the end tag

A few magic incantations are needed

- At the beginning of the file, to specify the version of XML used and the name of the file defining the syntax of your XML
  - XML can be used (in LHCb) for detector description or for event object description, and clearly the entities and their possible attributes are different !
  - The appropriate dictionary is copied in your package when configuring it, provided you have the proper **`requirements`** file, see later
  - This is used by **`emacs/xmleditor`** to propose entities and attributes

The rest is quite simple

- When you get used to it, of course !
- Looking at existing files is a good idea…

GOD and XML

- **GOD means Gaudi Object Description**
  - Gaudi product developed and maintained by Stefan Roiser
- **This is a "compiler"**
  - It converts the `xml` files to
    - C++ header files in the **`Event`** directory
    - C++ header and dictionary files in the **`dict`** dictionary
  - The dictionary files are compiled, and can be used by POOL when writing and reading those objects
    - Also used when interacting with them with Python, or in Panoramix
  - In fact the parsing is done by an open source product 'Xerces'
    - Uniform look and feel of all objects
    - Automatic generation of setters and getters
    - Standard format for **Doxygen** comments

- **An event model package has several features**
  - A **`requirements`** file with special content
    - Defines directories **`xml`**, **`Event`**, **`dict`** for special purposes
    - Declares GOD and what to do to generate header and dictionaries
    - new **`emacs`** generates a good requirement file when creating the file in a package whose name contains the string **Event**.
  - The source files are created in the **`xml`** directory
    - Plus possible implementation files in **`src`**, as usual.
      - ➔Event classes should usually have only simple methods, automatically generated by GOD
      - ➔Implementation files are rare, and discouraged.
    - One can have some short inlined code in the xml description.
      - ➔Careful with the reserved characters < > and &.
  - The Event directory is declared to be known to the compiler
    - So that the header file can be included in other packages.

◆The dictionary must be declared for containers

- For POOL, a **`KeyedContainer<plunk>`** has no relation to **`plunk`**, this is a completely independent object.
- A dictionary must be created not only for the object, but for containers thereof.
  - This is the reason for the magic incantation

    `&KeyedObject;`

    in the files describing objects inheriting from **`KeyedObject`**, that should be religiously copied when creating keyed objects.

Examples (TrStoredTrack)

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE gdd SYSTEM 'gdd.dtd'>
<gdd>
  <package name='TrEvent'>
    <class name     = 'TrStoredTrack'
           location = 'Rec/Tr/Best'
           id       = '10003'
           author   = 'Jeroen van Tilburg'
           desc     = 'An TrStoredTrack is a track which can be made persistent'>
      <location name="Velo" place="Rec/Tr/Velo"/>
      <location name="Seed" place="Rec/Tr/Seed"/>
      <location name="Match" place="Rec/Tr/Match"/>
      <location name="Forward" place="Rec/Tr/Forward"/>
      <location name="Follow" place="Rec/Tr/Follow"/>
      <location name="VeloTT" place="Rec/Tr/VeloTT"/>
      <location name="KsTrack" place="Rec/Tr/KsTrack"/>
      <base name='KeyedObject&lt;int&gt;'/>
      &KeyedObject;
      <attribute type='double'
                 name='charge'
                 desc='particle charge'/>
      <attribute type='int'
                 name='errorFlag'
                 desc='error flag'/>
```

Magic incantations

Default location

Other locations

Defines the base class, and the vectors/containers/…

Standard data member. Generate set and get methods

Defines bit fields in a word

```
<attribute type='bitfield'
          name='history'
          desc='origin of the track'>
  <bitfield name='unique'
            length='1'
            desc='Unique track (0=clone track, 1=unique track)'/>
  <bitfield name='velo‘ length='1' desc='Velo track'/>
  <bitfield name='seed' length='1' desc='Seed track'/>
  <bitfield name='match' length='1' desc='Matched track'/>
    (... )
  <bitfield name='ksTrack' length='1' desc='Ks decay track'/>
</attribute>
<relation type='TrStoredMeasurement'
          name='measurements'
          multiplicity='N'
          desc='SmartRefVector to TrStoredMeasurement'/>
<relation type='TrState'
          name='states'
          multiplicity='N'
          desc='SmartRefVector to TrStates'/>
<method type='const TrState* '
        name='closestState'
        argList='double zpos'
        const='TRUE'
        desc='get closest state to z position'/>
```

Vector of relations
to other objects

```
      <method type="double"
              name="lastChiSq"
              const="TRUE"
              desc="Get the last chi^2 of the track fit."/>
      <method virtual='TRUE'
              type='bool'
              name='isLong'
              const='TRUE'
              desc='Is the track a long track'>
      </method>
   </class>
  </package>
</gdd>
```

Close the elements

Recent news from GOD

- **New version v7r0 of GaudiObjDesc**
  - In the Gaudi release pipe-line, for v16r0.
  - Written in Python for easier management
    - Writing C++ from C++ is a pain !
  - Fixes several requests from RICH and Tracking
  - No change in the xml files nor in the resulting header files
    - Their functionality and generated names are identical
    - But better layout → better user readability.
      - ➔Better alignment
  - The number of lines of code in GaudiObjDesc has been reduced by a factor 5, allowing better management.

- **Thanks Stefan !**

Relations with Linker

What are Relations ?

◆This is a 'link' between two objects

- Tracks to MCParticles having produced them
- Tracks to Measurements used to build the track.

◆Some relations are structural

- Measurements are constituents of the track in some sense
- The relation exists **inside the object**.
  - They are indicated in xml by the `<relation />` entity.
  - Methods are generated by GOD to populate, copy, retrieve the set of relations
  - The implementation is by `SmartRef` or `SmartRefVector` according to the multiplicity of the relation
    - ➔One to one, or one to many
- This is handled by the Event Model.

◆Many relations are weaker

- Relation to MC truth is not available in real data.
  - Access to MC truth **is not part** of the structure of the event classes.
- One can specify something like "this object is related to that object"
  - And even qualify this relation with a weight.

◆Relations are independent objects

- This is implemented as a 'table'
  - Imagine an array with source, target and weight as rows…
- A tool is usually provided to use these relations
  - Named 'Associator'
- An algorithm is needed to create the relations
  - It runs usually when the source and target objects are all created
  - It can be complex, like the track associator to MCParticle

◆Original implementation by Vanya

- Set of highly templated classes, where source and target can be any object, and weight anything including an object.
  - Basically a vector of pairs of a source and a vector of pairs of a target and a weight.
- But the generation of dictionary (POOL requirement) for these classes requires some trick
  - Each possible relation has to be described.
  - Special package **`EventAssoc`** with one line of xml for each relation to store.

◆Speed is an issue

- When reading, as the relation table is sorted by the pointer to the objects, which are different upon reading.
  - Sorting is needed for fast access
  - Sorting each time an element is added is expensive
  - Sorting is not preserved when reading
    - ➔As the pointed objects are in different memory locations.

"Linker" implementation

◆Basic idea: Solve the previous problems !

- Of course there are limitations

◆Same table for all relations

- Represent objects by their container name and key
  - Works only with **`KeyedObject`** inserted in a container.
- But can also use a key independent of an object
  - Can link a channel ID to a MCParticle
    - ➔The relation is valid for digits, clusters, or internal representation in the L1 and Hlt packages
- Use the standard link table of any container to store the name of the containers to which there are relations
  - Source (and target) objects can be in several containers

◆Reading is fast

- There is no sorting at all.
- The table is just a collection of **`int`** and **`double`**.
  - Plus strings for the container names in the hidden features of a container.

◆Access is simplified

- There is no need for a tool with options
  - But Vanya's implementation can also be used this way, if we don't use the automatic invocation of the algorithm if the relation doesn't exist.
- A simple wrapper class do the job
  - This is created once per event, and answers with a simple syntax

- This relation can not be looked at as a STL container with iterators

How to use Linker relations ?

◆This is described in LHCb 2004-007

◆Creating a relation

```
#include "Linker/LinkerWithKey.h"
LinkerWithKey<TARGET,SOURCE> myLink( evtSvc(),
                                     msgSvc(), name );
```

SOURCE can be omitted if it the source inherits from **`KeyedObject<int>`**

'name' is a string, gives the location in the TES of the relation. One can use the same name as a container, as the name is prefixed with /link

```
myLink.link( source, target, weight = 1. );
```

- That's it…

Using a relation is as simple

```
#include “Linker/LinkedTo.h”
. . .
LinkedTo< MCParticle > myLink( evtSvc(), msgSvc(),
                        VeloClusterLocation::Default );
...
MCParticle* part = myLink.first( aCluster );
while ( NULL != part ) {
  bla = part->someMethod();
  ...
  weight = myLink.weight();
  ...
  part = myLink.next( );
}
```

The reverse relation can be retrieved also

- **`LinkedFrom`** instead of **`LinkedTo`**

This is in use since DC'04

- Needed to store the truth relations for L1 and HLT
  - Velo, IT and OT clusters by key.
- Also available for TrStoredTracks
  - Copy of the standard relations.