Analysis Model, Computing Model & Offline Software

Matthias Steinke
Ruhr-Universität Bochum

Panda Computing Model

- In 2008/2009 a group of Panda people worked out the *Panda Computing Model*
- The *Panda Analysis Model* became a part of the Computing Model
- The Analysis Model describes what the physics analysis users expect/need
- The rest of the CM lists requirements beside the physics analysis

My impression:

- Some points coming out of the CM were not tackled so far
- At some points the developments go into the wrong direction
- Concerning the actual discussion: The CM/AM tells us what the user interface of the software and the event data should look like

Object-Oriented Design

The Computing Model says (for good reasons):

> 7.2.3 Object-oriented design
>
> Object-oriented design, based primarily on the C++ programming language will be adapted for the PANDA software.

Two statements:

- We will make use of an OO design
  - we *always* have to think in objects
  - we always should make use of OO programming techniques
- We will use C++
  - the code has to be as C++ conform as possible
  - using C++ is *not* sufficient to have an OO design
  - we should use the features of C++ and the C++ standard lib

Event Data Structure

From the Analysis Model:

4.2.1 Parts of an analysis

In general, a physics analysis task can be divided into three steps:

- A fast preselection using the so called TAG event component that consists of brief summary data like particle multiplicities

4.3 Event data

The event model classifies the event data to different levels of detail.Three of these levels are relevant for a physics analysis:

- The **TAG** level contains brief event shape and event summary data ...

- The Analysis Object Data (**AOD**) hold reconstructed and composite particle candidates, lists of particle candidates, and objects directly connected with particle candidates ...

- The Event Summary Data **ESD** contains reconstruction objects which are not available in the AOD, but which are needed to refit tracks and to re-reconstruct photons.

Event Data Structure

- Remember: We have to think in objects!
- The main data component for physics analysis is the Analysis *Object* Data.
- There has to be a TAG component, and there has to an easy-to-use *infrastructure to apply TAG-based cuts* in a preselection step of an analysis job.

The offline software, therefore, has to allow in an easy and safe way to shift objects from AOD to ESD and vice versa, and to modify the TAG contents. A framework with a dynamic event structure would be advantageous for this.

The application framework has to support the switching of the event input from TAG only data to TAG+AOD data and to TAG+AOD+ESD data on an event-to-event basis ...

The storage of the event data has to be organized in a way that allows to export TAG+AOD only data to TIER1 and TIER2 centers for a physics analysis.

- Quite demanding requests
- A well designed structure of the event data seems to be a central point
  - An event data structure as described in the CM/AM has to be foreseen from the beginning!

Job Control

Job control as described in the CM/AM is still an open point:

> 4.5.1 Job control tools
>
> To run a physics analysis on a huge number of events it is necessary to split the task into a large number of jobs, each of them running on a subset of the events. To avoid that each analyst spends a part of her/his time with the creation, submission and error checking of a large number of jobs and log files, the analysis framework has to provide tools which automatically create the job files for all available events which fit to a selection mask given by the user.

- A common event store which holds all MC and beam events is needed
- An associated bookkeeping database is needed
- Tools to query the database and to generate job control files are needed

Analysis users think in events, selections of events, and particle candidates which they can combine and fit. They do not think in event data files, etc. The event I/O has to be hidden completely from them.

Skims

The Computing Model says:

> 4.5.1 Job control tools
>
> The TIER1 centers should host the parts of the analysis data which are relevant for the users and groups working mainly at the particular TIER1 in the form of so-called skims. Skims are prefiltered event collections, containing typically 1% or less of all events. The filter conditions are defined by the physics conveners in a way that one skim covers several similar analyses.

- The application framework has to provide the infrastructure for a skim production
  - the event I/O must be able to write multiple event output streams
  - the data model must be able to handle at least pointer skims, deep copy AOD skims, and deep copy ESD skims
- An easy-to-use user interface to define skim criteria is needed
- Tools to query the database and to generate job control files are needed

Adaptation of New Developments

The Computing Model says:

> 7.2.1 Design
>
> The complexity of the PANDA experiment requires that the software must be highly modular and robust. Furthermore it has to be flexible enough to meet the needs of the experiment throughout its design study and operational lifetime. ...
>
> 7.2.3 Object-oriented design
>
> Object-oriented design, based primarily on the C++ programming language will be adapted for the PANDA software.

But: C++ has no serialization! And that is definitely needed for event I/O

- HEP experiments invented their own serialization: part of Gaudi, Pool, Kanga, ...

The Computing Model also says:

> ... it is mandatory to follow on one hand well-established standards in the software development, such as the usage of ROOT and C++, but, on the other hand, to also look into more recent developments ...

The latest versions of the Boost library provide class serialization in C++

- Can we replace home-brewed code by parts of the (quasi) standard library?

Boost Serialization

The first paragraph of the Boost serialization library documentation:

> Here, we use the term "serialization" to mean the reversible deconstruction of an arbitrary set of C++ data structures to a sequence of bytes. Such a system can be used to reconstitute an equivalent structure in another program context. Depending on the context, this might used implement object persistence, remote parameter passing or other facility. In this system we use the term "archive" to refer to a specific rendering of this stream of bytes. This could be a file of binary data, text data, XML, or some other created by the user of this library.

A toy example to test the usability:

Event:
list of NeutCand*

NeutCand:
EmcBump*, pid

EmcBump:
list of EmcDigi*

EmcDigi:
theta, phi, energy

ToyEventWriteApp:
- create EmcDigis and EmcBumps
- create NeutCands from EmcBumps and store them on the neutral candidate list of an event
- store the event in an archive

ToyEventReadApp:
- read back the event from the archive
- dump the result to the screen

Serialization of a Toy Event

The first paragraph of the Boost serialization library documentation:

```
[matthias@pc12 ToyEvent]% ./ToyEventWriteApp
Event: 2 neutral candidates
neutral cand: theta = 43.02 phi = 16.2311 energy = 5.1345 pid = 1
EmcBump: theta = 43.02 phi = 16.2311 energy = 4.89
EmcDigi: theta = 42.2 phi = 16 energy = 0.88
EmcDigi: theta = 43.2 phi = 16 energy = 2.88
EmcDigi: theta = 43.2 phi = 17 energy = 1.13
neutral cand: theta = 44.1367 phi = 17 energy = 5.6385 pid = 2
EmcBump: theta = 44.1367 phi = 17 energy = 5.37
EmcDigi: theta = 43.2 phi = 17 energy = 1.13
EmcDigi: theta = 44.2 phi = 17 energy = 3.45
EmcDigi: theta = 45.2 phi = 17 energy = 0.79
```

```
[matthias@pc12 ToyEvent]% ./ToyEventReadApp
Event: 2 neutral candidates
neutral cand: theta = 43.02 phi = 16.2311 energy = 5.1345 pid = 1
EmcBump: theta = 43.02 phi = 16.2311 energy = 4.89
EmcDigi: theta = 42.2 phi = 16 energy = 0.88
EmcDigi: theta = 43.2 phi = 16 energy = 2.88
EmcDigi: theta = 43.2 phi = 17 energy = 1.13
neutral cand: theta = 44.1367 phi = 17 energy = 5.6385 pid = 2
EmcBump: theta = 44.1367 phi = 17 energy = 5.37
EmcDigi: theta = 43.2 phi = 17 energy = 1.13
EmcDigi: theta = 44.2 phi = 17 energy = 3.45
EmcDigi: theta = 45.2 phi = 17 energy = 0.79

neutral cand: theta = 43.02 phi = 16.2311 energy = 5.1345 pid = 1
EmcBump: theta = 43.02 phi = 16.2311 energy = 4.89
EmcDigi: theta = 42.2 phi = 16 energy = 0.88
...
```

Serialization of a Toy Event

The source code of the applications:

```
#include <cstddef>
#include <iomanip>
#include <iostream>
#include <fstream>
#include <string>
#include <list>
#include "EmcDigi.h"
#include "EmcBump.h"
#include "NeutCand.h"
#include "Event.h"

int main(int argc, char *argv[]) {
  Event event;
  EmcDigi digi1(42.2, 16.0, 0.88);  EmcDigi digi2(43.2, 16.0, 2.88);
  EmcDigi digi3(43.2, 17.0, 1.13);  EmcDigi digi4(44.2, 17.0, 3.45);
  EmcDigi digi5(45.2, 17.0, 0.79);
  EmcBump bump1;  EmcBump bump2;
  bump1.addDigi(digi1); bump1.addDigi(digi2); bump1.addDigi(digi3);
  bump2.addDigi(digi3); bump2.addDigi(digi4); bump2.addDigi(digi5);
  NeutCand cand1(bump1); cand1.setPid(NeutCand::gamma);
  NeutCand cand2(bump2); cand2.setPid(NeutCand::neutron);
  event.neutCands().push_back(&cand1);
  event.neutCands().push_back(&cand2);
  event.print();
  // make an archive
  std::ofstream ofs("toyEvents.txt");
  boost::archive::text_oarchive outputArchive(ofs);
  outputArchive << event;
  return 0;
}
```

```
#include <cstddef>
#include <iomanip>
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include "Event.h"
#include "NeutCand.h"
using namespace std;

int main(int argc, char *argv[])
{

  Event event;

  // open an archive
  std::ifstream ifs("toyEvents.txt");
  boost::archive::text_iarchive inputArchive(ifs);
  inputArchive >> event;

  event.print();
  std::cout << "\n\n";

// loop on neutral candidates
  for_each(event.neutCands().begin(), event.neutCands().end(),
mem_fun(&NeutCand::print));

  return 0;
}
```

Serialization of a Toy Event

The source code of the classes:

```
#include "Persistable.h"
#include "NeutCand.h"
#include <vector>

class Event {
  friend class boost::serialization::access;
  template<class Archive>
    void serialize(Archive & ar, const unsigned int /* file_version */) {
      ar & _neutCands;
      return;
    }
 public:
  Event();
  virtual ~Event();
  std::vector<NeutCand*>& neutCands();
  void print();
 private:
  std::vector<NeutCand*> _neutCands;
};
```

```
class EmcDigi {
    friend class boost::serialization::access;
    template<class Archive>
      void serialize(Archive & ar, const unsigned int /* file_version */) {
        ar & _theta; ar & _phi; ar & _energy;
        return;
      }
    double _theta;
    double _phi;
    double _energy;
```

```
class EmcBump {
  friend class boost::serialization::access;
  template<class Archive>
    void serialize(Archive & ar, const unsigned int /* file_version */) {
      ar & _digis;
      return;
    }

 public:
  EmcBump();
  virtual ~EmcBump();
  void addDigi(EmcDigi& digi);
  double theta();
  double phi();
  double energy();
  void print();
 private:
  void update();

  std::vector<EmcDigi*> _digis;
  bool _cacheValid;
  double _theta;
  double _phi;
  double _energy;
};
BOOST_CLASS_VERSION(EmcBump, 1)
```

Event Serialization with Boost

Btw, payload data gets stored *once* in the toy example:

```
double EmcBump::energy()
{
  if (!_cacheValid)
    update();
  return _energy;
}

void EmcBump::update()
{
  _theta = _phi = _energy = 0.0;

  if (_digis.size() > 0) {
    vector<EmcDigi*>::iterator digiIter;
    for (digiIter = _digis.begin(); digiIter != _digis.end(); digiIter++) {
      _theta += (*digiIter)->theta() * (*digiIter)->energy();
      _phi += (*digiIter)->phi() * (*digiIter)->energy();
      _energy += (*digiIter)->energy();
    }
    _theta /= _energy;
    _phi /= _energy;

    _cacheValid = true;
  }

  return;
}
```

The content of the text archive:

```
22 serialization::archive 5 0 0 0 0 2 0 2 1 0
0 3 1 1
1 0 0 3 0 5 1 1
2 42.200000000000003 16 0.88 5
3 43.200000000000003 16 2.8799999999999999 5
4 43.200000000000003 17 1.1299999999999999 1 2
5 3
6 3 0 5
4 5
7 44.200000000000003 17 3.4500000000000002 5
8 45.200000000000003 17 0.79000000000000004 2
```

- 5 EmcDigis got stored: Boost does **not** store the same object twice.
- Relations between objects get stored and restored *automatically*.
- Several archive types are available; for Panda events a streamlined archive should be developed.
- Using smart pointers for memory management and to realize load-on-demand would be obvious