Table of Contents

PAT User Data

Introduction

While the CMSSW data structure is very robust, it is impossible to foresee every possible use case for the data structure. With that in mind, users at the analysis level often want to add some more information to their objects that is relevant for their analysis, but not for others.

This is generally referred to as "user data". Many paradigms exist to incorporate user data, however the one adopted for the PAT is to include user data as a type of "mini-EDM" in the PAT objects themselves. This allows users to add their own data to their objects and retrieve them via a string associator.

Structure

The user data is implemented directly in PATObject. Generically, the user will add a piece of data to an object that inherits from `PATObject` (like `pat::Jet`, `pat::Muon`, etc). A string is passed along with the data, which can be used to access the information later. The objects are stored as two vectors that are maintained simultaneously:

```
/// User data object
       std::vector<std::string>      userDataLabels_;
       pat::UserDataCollection       userDataObjects_;
       // User float values
       std::vector<std::string>      userFloatLabels_;
       std::vector<float>            userFloats_;
       // User int values
       std::vector<std::string>      userIntLabels_;
       std::vector<int32_t>          userInts_;
```

Examples

addUserFunction

The most common usage is to reconstruct the variables within the `PATObject` in question, without much external input. This can be done with the StringCutParser directly in the `PATObject` producer itself with the `addUserFunction` method. The producer will add a user float to the `userFloat` vector with the names that the user provides. For instance, to add the `relIso` variable to `pat::Muon`, one could do:

```
# Add user data to the object itself.
  process.patMuons.userData.userFunctions = cms.vstring('((trackIso+caloIso)/pt)')
  process.patMuons.userData.userFunctionLabels = cms.vstring('relIso')

  # Do a selection on relIso
  process.selectedPatMuons.cut = cms.string('pt > 20. & abs(eta) < 2.1 & userFloat("relIso") < 0.
```

This will place only the muons that pass the *pt*, *eta*, and *relIso* cuts into the `selectedPatMuons` vector.

The user can then access the data in `FWLite`, for instance, as

```
Handle<vector<pat::Muon> > h_muons;
   iEvent.getByLabel( h_muons, "selectedPatMuons");
   if ( h_muons->size() > 0 ) {
     pat::Muon const & myMuon = h_muons->at(0);
     if ( myMuon.hasUserData("relIso") == false ) break;
     double relIso = myMuon.userFloat("relIso");
     my_histogram->Fill( relIso );
```

```
}
```

`addUserFloat` : Dedicated `EDProducer`

A more general case is when users wants to create their own `EDProducer` to do some non-trivial computation that cannot be handled via the StringParser. In this case, the user can create an `EDProducer` as normal (make a "produces" method, then "put" the modified objects into the event), and access the `addUserFloat` method directly:

```
pat::Muon * someMuon = getAMuonFromSomewhere.clone();
   float relIso = (someMuon->caloIso() + someMuon->trackIso() / someMuon->pt();
   someMuon->addUserFloat( "relIso", relIso);
```

To access them, one does exactly what was described above. As usual, the users must put their modified objects into the event. A concrete example of an `EDProducer` storing `UserFloats` into a pat::Candidate collection can be found in this Z $\mu^+\mu^-$ analysis code.

Variables stored as `UserFloats` can be get into an EDM Ntuple by using the CandViewNtpProducer tool.

To add user "floats" the User has to create the edm::ValueMap of the floats to the object which are used to create pat::Objects. Below is the example of how to add cosmic id variables as user floats to the *pat::Muons*.

In the configuration file:

show config part Hide config part
a) First include the produces if the value maps:

```
process.cosmicCompatibility = cms.EDProducer("CosmicID",
                                              src=cms.InputTag("cosmicsVeto"),
                                              result = cms.string("cosmicCompatibility")
                                              )
process.timeCompatibility = process.cosmicCompatibility.clone(result = 'timeCompatibility')
process.backToBackCompatibility = process.cosmicCompatibility.clone(result = 'backToBackCompatibi
process.overlapCompatibility = process.cosmicCompatibility.clone(result = 'overlapCompatibility')
```

Then load the pat sequences and modify the pat muon producer:

```
# load the standard PAT config
process.load("CMS.PhysicsTools.PatAlgos.patSequences_cff")
#Add user defined variables for muon
process.patMuons.userData.userFloats.src = ['cosmicCompatibility','timeCompatibility','backToBack
```

Finally do not forget to include everything you want to run in the path:

```
process.pat_step = cms.Path(
    process.cosmicCompatibility +
    process.timeCompatibility +
    process.backToBackCompatibility +
    process.overlapCompatibility *
    process.patDefaultSequence
)
```

The next step is to write the producer of the *edm::ValueMap*, an example is included below:

show producer Hide producer

```
// system include files
#include <memory>
#include <iostream>
#include <string>
```

```
#include <vector>

// user include files
#include "FWCore/Framework/interface/Frameworkfwd.h"
#include "FWCore/Framework/interface/EDProducer.h"

#include "FWCore/Framework/interface/Event.h"
#include "FWCore/Framework/interface/MakerMacros.h"

#include "FWCore/ParameterSet/interface/ParameterSet.h"
#include "DataFormats/Common/interface/ValueMap.h"

#include "DataFormats/MuonReco/interface/MuonFwd.h"
#include "DataFormats/MuonReco/interface/Muon.h"
#include "DataFormats/MuonReco/interface/MuonCosmicCompatibility.h"

class CosmicID : public edm::EDProducer {
   public:
      explicit CosmicID(const edm::ParameterSet&);
      ~CosmicID();

   private:
      virtual void beginJob() ;
      virtual void produce(edm::Event&, const edm::EventSetup&);
      virtual void endJob() ;

      // ----------member data ---------------------------
        edm::InputTag src_;
        std::string result_;
};

CosmicID::CosmicID(const edm::ParameterSet& iConfig)
{
        src_= iConfig.getParameter<edm::InputTag>("src");
        result_ = iConfig.getParameter<std::string>("result");
        produces<edm::ValueMap<float> >().setBranchAlias("CosmicDiscriminators");
}


CosmicID::~CosmicID()
{

}

// ------------ method called to produce the data  ------------
void
CosmicID::produce(edm::Event& iEvent, const edm::EventSetup& iSetup)
{
        using namespace edm;
        using namespace reco;
        Handle<edm::ValueMap<reco::MuonCosmicCompatibility> > CosmicMap;
        iEvent.getByLabel( src_, CosmicMap );
        edm::Handle<reco::MuonCollection> muons;
        iEvent.getByLabel("muons",muons);
        std::vector<float> values;
        values.reserve(muons->size());

        unsigned int muonIdx = 0;
        for(reco::MuonCollection::const_iterator muon = muons->begin();
                        muon != muons->end(); ++muon) {
                reco::MuonRef muonRef(muons, muonIdx);
                reco::MuonCosmicCompatibility muonCosmicCompatibility = (*CosmicMap)[muonRef];

                if(result_ == "cosmicCompatibility") values.push_back(muonCosmicCompatibility.cos
                if(result_ == "timeCompatibility") values.push_back(muonCosmicCompatibility.timeC
                if(result_ == "backToBackCompatibility") values.push_back(muonCosmicCompatibility
                if(result_ == "overlapCompatibility") values.push_back(muonCosmicCompatibility.ov
```

```
                ++muonIdx;
        }

        std::auto_ptr<edm::ValueMap<float> > out(new edm::ValueMap<float>());
        edm::ValueMap<float>::Filler filler(*out);
        filler.insert(muons, values.begin(), values.end());
        filler.fill();

  // put value map into event
        iEvent.put(out);
}

// ------------ method called once each job just before starting event loop  ------------
void
CosmicID::beginJob()
{
}

// ------------ method called once each job just after ending the event loop  ------------
void
CosmicID::endJob() {
}

//define this as a plug-in
DEFINE_FWK_MODULE(CosmicID);
```

Finally access your variables in the analyzer:

show analyzer part Hide analyzer part

```
iEvent.getByLabel(m_muoLabel_,m_PAT_muonHandle);
edm::View<pat::Muon> pat_muons = *m_PAT_muonHandle;
 for( edm::View<pat::Muon>::const_iterator muon_iter = pat_muons.begin();
                        muon_iter !=pat_muons.end(); ++muon_iter)
        {
          float cosmicCompatibility= muon_iter->userFloat("cosmicCompatibility");
          float timeCompatibility = muon_iter->userFloat("timeCompatibility");
          float backToBackCompatibility = muon_iter->userFloat("backToBackCompatibility");
          float overlapCompatibility = muon_iter->userFloat("overlapCompatibility");
}
```

`addUserData` : Dedicated `EDProducer`

It is possible to add **any** class to a `PATObject`, however there is a special implementation to add floats and ints directly, since they are envisioned to be the most common use cases. Therefore they are stored in their own vectors directly for performance reasons.

To add a generic object, be aware that a dictionary library must exist for the object in question. If users are making their own generic classes to add, these users will need to include a library for the object somewhere.

To add the information, one does

```
pat::Muon * someMuon = getAMuonFromSomewhere.clone();
   UsersFavoriteData * usersData = getYourUserDataFromSomewhere();
   someMuon->addUserData<UsersFavoriteData>( "myData", *usersData );
```

As usual, the users must put their modified objects into the event. To access the information, one does

```
pat::Muon const & myMuon = getMyMuonFromSomewhere();
   if ( myMuon.hasUserData("myData") == false ) break;
   UsersFavoriteData const & usersData = myMuon.userData<UsersFavoriteData>("myData");
```

`pat::CompositeCandidate` Example

The `pat::CompositeCandidate` is very simple. It just adds the functionality from the `pat::PATObject` to the `reco::CompositeCandidate`. So adding user data to a `pat::CompositeCandidate` is quite trivial.

A snippet of an example is:

```
pat::CompositeCandidate jpsi;
            jpsi.addDaughter( *imuon );
            jpsi.addDaughter( *jmuon );

            AddFourMomenta addp4;
            addp4.set( jpsi );

    double dR = reco::deltaR<pat::Muon,pat::Muon>( *imuon, *jmuon );

            jpsi.addUserFloat( "dR", dR );

    if ( fabs( jpsi.mass() - JPSI_MASS ) < 1.0 ) {
              jpsiCands->push_back( jpsi );
            }
```

The access of the "dR" variable in FWLite is trivial:

```
// fwlite::Handle to to jpsi collection
    fwlite::Handle<std::vector<pat::CompositeCandidate> > jpsis;
    jpsis.getByLabel(event, "patJPsiCandidates");

    // loop jpsi collection and fill histograms
    for(unsigned i=0; i<jpsis->size(); ++i){
      cout << "jpsi " << i << ", mass = " << jpsis->at(i).mass() << ", dR = " << jpsis->at(i).use
    }
```

An example of how to add user data can be found in Z $\mu^+\mu^-$ analysis code.

Contacts

- Salvatore Rappoccio (salvatore.rappoccio@cernSPAMNOTch)

Review status

Reviewer/Editor and Date (copy from screen)	Comments
SalvatoreRoccoRappoccio - 23 Jan 2007	created template page

Responsible: ResponsibleIndividual
Last reviewed by: Most recent reviewer

This topic: CMSPublic > SWGuidePATUserData
Topic revision: r11 - 2011-12-05 - SudhirMalik