D3PD Maker Package

Scott Snyder

BNL

Jan 18, 2010

Design goals

- Focus only on making D3PDs from EDM objects in StoreGate. Don't try to build an analysis framework.
  Assumption is that any needed analysis will be done in upstream Athena algorithms, and the results stored in SG (and can then also be written to D[12]PDs.)
- Focus on making flat tuples that can be rapidly read in root with no extra dictionaries. But don't preclude embedding objects as well.
- Variables in the tuple should be grouped into related blocks at a relatively fine level of granularity. Each block is associated with a 'level of detail;' blocks can be selected by choosing a level or detail or individually.
- The tuple should be easily extensible by user code without requiring changes to the core packages.

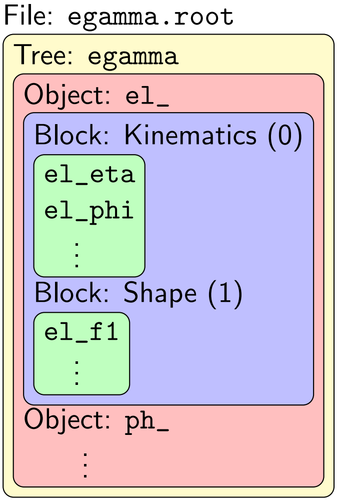
Definitions

- Files contain trees.
- Trees contain objects.

  Object ≡ an object or container from SG. Identified by a variable prefix.
- Objects contain blocks.

  Block ≡ group of related variables for an object. Defines granularity of variables (level-of-detail).
- Blocks contain variables.

  Any type known to Root.



Using D3PDs

- D3PDMaker described here: `https://twiki.cern.ch/twiki/bin/view/AtlasProtected/D3PDMaker`
- Pointers to some existing D3PDs are given on that page.
- D3PDs are standard Root tuples. Those made so far use only simple types and vectors of simple types.
- Egamma tuples contain detailed information about electrons, photons, and muons, and basic kinematic information about other objects.
- Physics tuples contain more information about jets, tracks, clusters, and missing E_T.
- Other groups have defined other tuples.
- Lists of variables for electrons, photons, and muons may be found in `D3PDMakerConfig/doc`.

Available tuple objects

- egamma
  - Contains most of the requested variables: Kinematics, shower shape, track match, conversions, egamma trigger decisions, truth and trigger matches.
  - Also flags for passing loose/medium/tight selections.
- Muons.
  - No trigger or truth matching yet.
- Taus (Dugan O'Neil).
- Tracks/vertices/clusters (Maarten Boonekamp).
- Jets (Pier-Olivier Deviveiros).
- Missing E_T (Jet Goodson).
- Truth (M. Boonekamp, S. Snyder)
- In progress:
  - Full tringger information (Attila K.)

Making D3PDs

- Current list of tags for making D3PDs is here: `https://twiki.cern.ch/twiki/bin/view/AtlasProtected/D3PDMaker`
  - Last version is Dec. 18.
  - Tested with 15.6.1. Most of these tags should be in 15.6.3.
  - In 15.6.X and 16.X nightlies.
- Example top-level JO: `D3PDMakerConfig/AODToEgammaD3PD.py`.
- Edit to specify your input and output files.
- Some options:

```
from D3PDMakerConfig.D3PDMakerFlags import D3PDMakerFlags
D3PDMakerFlags.DoTrigger = True # Default
from RecExConfig.RecFlags import rec
rec.DoTruth = True # Autoconfigured by default.
# Set SG key.  First listed key that exists is used.
D3PDMakerFlags.ElectronSGKey = \
  'ElectronAODCollection,ElectronCollection'
```

Basic configuration

- Tuple configured with lines like this:

```
from D3PDMakerConfig.egammaD3PD import egammaD3PD
alg = egammaD3PD ('egamma.root',  # Filename
                  'egamma',       # Treename
                  10)       # Desired level of detail.
```

- Also exists `physicsD3PD`.
- Can add user-defined objects to the tuple:

```
from egammaD3PDMaker.ElectronD3PDObject \
       import ElectronD3PDObject
alg += ElectronD3PDObject (10, sgkey = 'my_electrons',
                           prefix = 'myel_')
```

Extending the D3PD Maker

Tuple making

- Each tree is made by an instance of MakerAlg:

```
alg = D3PDMakerCoreComps.MakerAlg(
          'testTree',
           topSequence,
           file = 'out.root')
```

- Multiple trees can be created in the same file.
- Root dependencies are isolated behind abstract interfaces: ID3PDSvc creates an instance of ID3PD.
- Specific implementations exist for Root (RootD3PDSvc, RootD3PD).
- In principle, can write other forms of tuple (HDF5, UserData, etc.) by replacing this service.

Object filling

- Each tree has a list of `IObjFillerTool` instances.
- Each filler tool:
  - Retrieves an input object.
  - If it's a collection, iterate over the contents.
  - Iterate over list of block filler tools.
- Tools available include:
  - `ObjFillerTool` — for single objects.
  - `VectorFillerTool` — for collections. Puts the results in `std::vector` objects in the tuple.
- These tools are generic (don't depend on the type of object being processed).
- Take "Getter" tools as properties. Abstracts the retrieval of the input objects. Usually one of:
  - `SGObjGetterTool`
  - `SGDataVectorGetterTool`

  to retrieve a single object or a collection from StoreGate.

Object filler configuration example

- Define electron object:

```
from D3PDMakerCoreComps.D3PDObject \
   import make_SGDataVector_D3PDObject
from D3PDMakerConfig.D3PDMakerFlags import D3PDMakerFlags

ElectronD3PDObject = make_SGDataVector_D3PDObject \
  ('ElectronContainer',
   D3PDMakerFlags.ElectronSGKey(),
   'el_')
```

- Think of this like a class, which you instantiate by adding to a tuple:

```
    alg += ElectronD3PDObject (level_of_detail)
    alg += ElectronD3PDObject (level_of_detail,
                               sgkey = 'ReprocessedEle',
                               prefix = 'reel_')
```

Block filler tools

- Takes an object and copies some data from it to the tuple.
- Define the granularity at which the contents of the tuple may be controlled.
- Always operates on a single object; iterating over objects in a collection and putting the results in a container is the responsibility of the object filler tool.
- Defined by interface `IBlockFillerTool`.
- User code will generally derive from the type-safe wrapper `BlockFillerTool<T>`.
- Class objects can be added to the tuple as well as simple types, using the same interface.
- Block filler tools are attached to D3PD objects. Each has a name and a level of detail.

```
ElectronD3PDObject.defineBlock (0, 'Kinematics',
                                FourMomFillerTool)
```

Extending standard objects

- You can customize your tuple by adding new block filler tools to standard objects.

```
from D3PDMakerConfig.egammaD3PD import egammaD3PD
from egammaD3PDMaker.ElectronD3PDObject \
       import ElectronD3PDObject
from mytools import mytoolsConf

# All electron objects will have this block.
ElectronD3PDObject.defineBlock (1, 'myblock1',
                                mytoolsConf.myblock1)
alg = egammaD3PD (tupleOutputFile)

# Electron objects added from here on will have this block.
ElectronD3PDObject.defineBlock (1, 'myblock2',
                                mytoolsConf.myblock2)
alg += ElectronD3PDObject (10, sgkey = 'my_electrons',
                           prefix = 'myel_')
```

Example: FourMomFiller

```
class FourMomFillerTool
  : public BlockFillerTool<INavigable4Momentum>
{
public:
  FourMomFillerTool (const std::string& type,
                     const std::string& name,
                     const IInterface* parent);
  virtual StatusCode book();
  virtual StatusCode fill (const INavigable4Momentum& p);

private:
  float* m_pt;
  float* m_eta;
  float* m_phi;
};
```

Example: FourMomFiller

```
StatusCode FourMomFillerTool::book()
{
  CHECK( addVariable ("pt",  m_pt)  );
  CHECK( addVariable ("eta", m_eta) );
  CHECK( addVariable ("phi", m_phi) );
  return StatusCode::SUCCESS;
}


StatusCode
FourMomFillerTool::fill (const INavigable4Momentum& p)
{
  *m_pt  = p.pt();
  *m_eta = p.eta();
  *m_phi = p.phi();
  return StatusCode::SUCCESS;
}
```

Tuple configuration

- So can configure an entire tuple with a function like this:

```
def makeTestD3PD (file, level = 10,
                  tuplename = 'test', seq = topSequence,
                  D3PDSvc = 'D3PD::RootD3PDSvc'):

    alg = D3PDMaker.MakerAlg(tuplename, seq, file = file,
                             D3PDSvc = D3PDSvc)
    alg += EventInfoD3PDObject (level)
    alg += ElectronD3PDObject (level)
    alg += JetD3PDObject (level)
    return alg
```

- Usage from top JO:

```
alg = makeTestD3PD ('out.root')
alg += ElectronD3PDObject (10, prefix = 'myel_',
                           sgkey = 'MyElectrons')
```

Type conversion caveat

- The D3PD maker may need to convert pointer types.
  - Example: You use a getter returning an ElectronContainer and a block filler tool expecting a pointer to an INavigable4Momentum.
- Type conversion is done using SGTools/BaseInfo.h.
- Information for this comes from DATAVECTOR_BASE and SG_BASE declarations in the headers. (See DataVectorMacros twiki page.)
- If this is not present, then the type conversions may fail.
- (May also fail if no reflex dictionary exists for the involved types. This should be improved in the 16.X nightlies.)
- If the appropriate macros are not present in the header files, the information can be added in D3PD packages using SG_ADD_BASE:

```
#include "EventKernel/INavigable4Momentum.h"
#include "EventKernel/IParticle.h"
#include "SGTools/BaseInfo.h"
SG_ADD_BASE (IParticle, SG_VIRTUAL(INavigable4Momentum));
```

- These should eventually be migrated back to EDM classes, however.

More on block configuration

- Besides specifying a simple level-of-detail, one can also request that specific blocks be included or excluded:

```
alg += ElectronD3PDObject (1,
                           include = ['RefittedTrk'],
                           exclude = ['HadLeakage'])
```

- You can also pass arguments to specific block fillers:

```
alg += ElectronD3PDObject (1,
                           {'Kinematics' :
                              {'WriteE' : False}})
```

  or equivalently:

```
alg += ElectronD3PDObject (1, Kinematics_WriteE = False)
```

More on block configuration

- Each block as a specified level-of-detail:

```
d3pdobject.defineBlock (1, 'MyBlock', MyFiller)
```

- Level-of-detail can now be a function. Gets as arguments the requested level-of-detail and the block filler arguments (may modify).

```
def my_lod_func (reqlev, args):
  if reqlev < 1: return False
  if reqlev >= 2:
    args['IncludeMore'] = True
  if reqlev >= 3:
    args['IncludeEvenMore'] = True
  return True
d3pdobject.defineBlock (my_lod_func, 'MyBlock', MyFiller)
```

UserData

- Doing non-trivial analysis tasks in the block filler code is discouraged; this should instead be done in separate Athena algorithms.
- Results can be written to new EDM classes and put in StoreGate.
- Sometimes, making new EDM classes is overkill.
- Can instead use `UserDataSvc` to associate arbitrary objects ("decorations") with EDM objects.
- These can be saved in pool files.
- There is also a generic tool to save these into D3PD:

```
ElectronD3PDObject.defineBlock \
      (1, 'UDLayer1Shape',
       D3PDMakerCoreComps.UserDataFillerTool,
       # Prefix to add in front of UD labels.
       UDPrefix = D3PDMakerFlags.EgammaUserDataPrefix(),
       #       d3pdvar       ud label       type
       #                     [null->d3pdvar]
       Vars = ['deltaEmax2', '',            'float' ] )
```

UserData

- This is now used for a few egamma variables that were too complicated to calculate in the filler tools.
- egammaD3PDAnalysis contains Athena algorithms that calculate the variables and store them as UserData.
- egammaD3PDMaker then stores these variables into the d3pd from UserData.
- Example: Conversion truth analysis. (Algorithm adapted from PhotonAnalysisUtils.)

Simple associations

- Simple (single) associator tool maps from one object to another.
- Example:
    - Have a tool that fills track parameters `d0`, etc. from a `TrackParticle`.
    - Write association tools:
        - `egamma` → `TrackParticle`
        - `Muon` → `TrackParticle`
    - Can then use the common `TrackParticle` tool with both.
    - Configuration:

```
ElectronTPAssoc = SimpleAssociation \
   (ElectronD3PDObject, egTPAssocTool,
    prefix = 'track_', matched = 'matched')
ElectronTPAssoc.defineBlock (1, 'Track', TPParamsFillerTool)
```

- Makes variables `el_track_d0`, etc.
- Also makes flag variable `el_track_matched`.

Simple association tool example

Such tools should generally derive from `SingleAssociationTool`.

```
class egTPAssocTool
  : public SingleAssociationTool<egamma, Rec::TrackParticle>
{
public:
  egTPAssocTool (const std::string& type,
                 const std::string& name,
                 const IInterface* parent);

  virtual const Rec::TrackParticle* get (const egamma& p)
  { return p.trackParticle(); }
};
```

Other available association filler tools

- Associations can either be single (as just shown) or multiple (in which a single source object associates to a set of target objets).
- Note the separation between the formation of an association and how the association is represented in the tuple.
- `IndexAssociation` — Represent a single association by an index into a collection. Need to reference a Getter to define the collection within which to index.
- `IndexMultiAssociation` — Represent a multiple association by a vector of indices into a collection.
- `ContainedVectorMultiAssociation` — Represent a multiple association by putting result variables into vector. This will usually create nested vectors.
- `ContainedMultiAssociation` — Represent a multiple association by repeating the tuple rows.

Metadata

- Metadata may be associated with a d3pd tree; each tree has an associated set of (key, value) pairs.
- Implementation:
  - For a tree named “Foo” we create a root directory “FooMeta”. The objects in this directory are the metadata; their names are the keys.
- Straightforward to read in interactive root (at least if a `TObject` type is used). Metadata need not be read until needed.
- `ID3PD` object gets a new method:

```
TObjString* obj = new TObjString ("some string");
StatusCode sc = d3pd->addMetadata ("mdkey", obj);
```

- Interface `IMetadataTool` defines:

```
virtual StatusCode writeMetadata (ID3PD* d3pd) = 0;
```

- `D3PD::MakerAlg` takes a list of these tools, called at end-of-job.

```
alg.MetadataTools += [LBMetadataConfig()]
```

Metadata

- `LBMetadataTool` writes the luminosity XML string to the d3pd at the end of the job.
- Using metadata key `Lumi`.
- Configured by default for the egamma and physics D3PDs.

Summary

- D3PDMaker is available for people to try out.
- For more information see: `https://twiki.cern.ch/twiki/bin/view/AtlasProtected/D3PDMaker` and files in `D3PDMakerConfig/doc`.
- Some standard tuple configurations available, or build your own.
- Production of standard D3PDs under discussion.
- Tools for automatic validation are under construction.