TheNtupleMaker: Automated Ntuple Creation from EDM Data

Harrison B. Prosper[1] *and Sezen Sekmen*[2]
[1]Department of Physics, Florida State University, FL 32306, USA
[2]Department of Physics, Kyungpook National University, Korea

Abstract

TheNtupleMaker is an automated system for creating ntuples from data in the EDM data formats of the CMS Collaboration. In particular, TheNtupleMaker (TNM for short) works on any CMS Open Data miniAOD files that depend on ROOT 5. The system, which uses the ROOT 5 reflex mechanism, provides a simple Python-based configuration file to specify the EDM data to be extracted and also generates an analyzer that can be used to read and analyze the resulting ntuples. Furthermore, TNM provides two mechanisms to add user-defined variables to the ntuples and implements a mechanism to skim events. The system is self-documenting and stores, in the ntuple, the most useful provenance information.

Contents

1 Introduction

1.1 Motivation

Physicists tend to be pragmatic individuals who just want to get the job done. Therefore, given the choice between two analysis systems, most physicists will opt for the system that is likely to yield the shorter path to a result. Experience at the Tevatron and the LHC indicates that the shorter path is almost always the same as the path requiring the least amount of technical computing skill; hence, the ubiquitous use of `ROOT` *ntuples* in analyses. Ntuples are used not only because they are simple, but also because they require the installation of a minimal amount of software infrastructure, namely, `ROOT`. Anyone with a modicum of computing skills can participate in an analysis and make an intellectual contribution to it. An excellent example of this truism is the successful fourteen year search for single top quark production by the D0 Collaboration [1]. By any measure, the D0 single top analyses were very sophisticated; yet almost everything was done using ntuples and the fraction of group members who made substantial contributions to the analyses was atypically high.

A large number of ntuple-makers exist in CMS and no doubt in every LHC collaboration, and a large number existed in CDF and D0. Even within a given CMS Physics Analysis Group (PAG) multiple ntuple-makers exist. (Note however that in ATLAS, a single and centrally maintained ntuple-making system exists. Using this single system, ntuples can be custom designed according to the common needs of groups of physics analysis teams, and hence, a lot of time, effort and computing power is saved.)

Given that many ntuple-makers already exist in CMS, an obvious question is: why on earth would one contemplate (in 2009 when the development of `TheNtupleMaker` began) yet another ntuple-maker? As the experience at D0 showed, the availability of a simple analysis infrastructure is a necessary condition for empowering physicists regardless of their computing expertise. But it is not sufficient. The sufficient condition is that everyone within a group should have access to the *same* analysis infrastructure. Curiously, this obvious sufficient condition is routinely ignored [1]. This results in an immense time loss and inefficiency since analysts constantly have to decipher and adapt to new analysis infrastructures as they move to new analyses and new collaborations [2]. Having many analysis infrastructures also reduces the amount of time and person-power that can be devoted to validate each of these infrastructures thoroughly.

The chief motivation for embarking on the `TNM` project in December 2009 was the absence, at the time, of a *simple* and *standard* infrastructure for making ntuples with a *standard format* from events in CMS Event Data Model (EDM) formats, such as `RECO`, `AOD`, `PAT`, `CMG`, etc. Since all these formats are based on EDM, it was somewhat surprising that no such central tool existed. Our goal was to fill this void with a tool, `TNM`, for making ntuples automatically, where the content of the ntuple is straightforwardly defined by the user (and which additionally generates an analyzer to read and analyze the resulting ntuples). Furthermore, `TNM` provides mechanisms to add user-defined variables to the ntuples and skim events. `TNM` is a *self-documenting* tool, which automatically tracks key provenance information pertaining to the EDM objects that were used in making the ntuples.

The other motivation for embarking on this journey was to explore an answer to a "What if?" question: what benefits might accrue if our field made a concerted effort to take seriously user-friendly interfaces? What if we were able to empower a much larger pool of creative people to engage in what we do; how much more progress might we make?

This document is structured as follows. We first summarize the design principles of `TNM` in Section 1.2, then briefly explain the implementation of the design in Section 1.3. This will be followed in Section 2.1 by a detailed description of `TNM` usage, a note on provenance in Section 3.6, and some

[1] During the run up to the summer 2011 conference season, one CMS group was reduced to relying almost solely on one or two post docs to do all the work, leaving dozens of others unable to help, because of the difficulties with the analysis infrastructure.

[2] One of the authors suffered from this to the point of extreme frustration.

conclusions. Readers who would like to start using TNM immediately can proceed directly to Section 2.1, though we recommend first reading the documentation at https://github.com/hbprosper/TheNtupleMaker. Technical details, which have been skipped in order not to appall the non-geeks among us, can be found in the Appendices.

1.2 Design principles

The design principle of TNM is guided by the following advice:

Things should be made as simple as possible, but not any simpler.
Albert Einstein

The ROOT[3] package from CERN has evolved into an impressively complete system that is able, for example, to store user-defined objects in ROOT files, a feature that is exploited by the CMS EDM. However, this object-oriented view is sometimes at odds with how physicists view data. For many physicists, an event is simply an ntuple of variables whose contents may change as the need arises. The point is not that objects are useless—on the contrary, they can be very useful; rather the point is that for a given analysis we typically make use of only a small subset of the attributes of the event objects.

Consider, for example, the C++ class pat::Electron from CMSSW, the codebase of the CMS Collaboration. If you unfold the inheritance hierarchy of this class, you would find that pat::Electron objects export about 150 *simple* methods! We call a method simple if its return type is a *fundamental type*, either a double, int, unsigned int, float, or bool. Here is an example of a simple method:

```
float iso = electron.trackIso(),
```

where electron is a pat::Electron object. It is a very unusual analysis indeed that needs every one of these 150 methods. Moreover, if you unfold the object one more level, to a *compound* method, the number of methods increases by more than an order of magnitude. A *compound* method is one that entails indirection. Here is an example:

```
double pt = electron.gsfTrack()->pt().
```

Some compound methods go one level deeper. Here is one from GenEventInfoProduct

```
double pdf1 = object.pdf()->xPDF.first,
```

which returns the value of the parton distribution function (PDF) for the first parton. When the classes in the CMSSW subsystems DataFormats and SimDataFormats are unfolded to a compound method depth of two, this yields more than 50,000 methods. Of course, only a fraction of these are actually useful for analysis. Even so, this amounts to of order 5,000 methods.

The above considerations suggest the following simplified programming model for particle physics data and CMS data, in particular: they are a large collection of floats, doubles, ints, unsigned ints, or bools each of which is associated with a simple or compound method. The purpose of TNM is to provide systematic access to any subset of these methods and create an ntuple from the subset. Inspired by Einstein's advice, we adhere to the following design principles:

1. The resulting ntuple should be of the simplest possible format.
2. The configuration file should be self-documenting.
3. The system should have an automatic way to construct the ntuple branch names.
4. The system should be flexible enough to deal with indirection.
5. The system should permit the addition of user-defined methods to the existing pool of methods.
6. The system should not try to do more than it should.

The following design choices are consistent with these design principles:

[3] http://root.cern.ch

1. The ntuple uses ROOT's variable-length array mechanism, rather than STL vectors[4]. Instead the mapping to vectors is done at runtime when an ntuple is read.
2. The configuration file uses the *exact* name and return type of the method to be accessed.
3. The branch name is a specific concatenation of the class name (see below), the label used by getByLabel[5], and the method name.
4. TNM imposes a maximum compound method indirection depth of 20[6].
5. The addition of user-defined methods is handled using a "helper" mechanism (see Section 3.2). In addition, variables can be added directly to the ntuple using a "macro" mechanism (see Section 3.3), which can also be used to save specific objects and specific events. However, ...
6. ... we resist the temptation to make a paintbrush that also fries eggs! The purpose of TNM is very specific: it is to map methods to numbers and store the latter, event by event. We also resist the temptation (which is easy!) to build a compiler. Therefore, we impose the following restrictions:
   (a) Simple and compound methods must return a *numerical* fundamental type.
   (b) The only arguments allowed are *none* or any combination of *values* of fundamental types and strings.

1.3 Implementation

In this section, we give a general description of the implementation of TNM. Technical details are provided in the Appendices.

TNM is an EDAnalyzer

```
class TheNtupleMaker : public edm::EDAnalyzer
{
public:
  explicit TheNtupleMaker(const edm::ParameterSet&);
  ~TheNtupleMaker();
private:
  virtual void beginJob();
  virtual void beginRun(const edm::Run&, const edm::EventSetup&);
  virtual void analyze(const edm::Event&, const edm::EventSetup&);
  virtual void endJob();
    : :
};
```

that is called by cmsRun, the standard tool in CMS to run programs, and controlled by a configuration file, which specifies the variables to be written to the ntuple. Its principle features are:

- A mechanism to call an arbitrary collection of simple and compound methods that return fundamental types, have no arguments or have arguments comprising values of fundamental types and strings.
- A *helper* mechanism to permit the addition of user-defined methods to the pool of existing methods. Typically, these methods are needed when a desired number is a complicated function of the existing methods.
- A *macro* mechanism to permit the addition of user-defined variables to the ntuple directly.

The pseudocode below presents a high-level (conceptual) view of TNM.

[4] In the ROOT 6 version, under development, we plan, however, to use STL vectors in order to provide more flexibility than is provided in the current version of TheNtupleMaker.

[5] In the current version of CMSSW this has been replaced by another function.

[6] This ought to be sufficient!

```
constructor
  (1) output = createEmptyNtuple()
  (2) config = decodeConfigurationFile()
beginRun
  if first run:
  (3) initializeHLTConfiguration()
  (4) buffers = createBuffers(config)
  (5) output.createBranches(buffers)
analyze
  (6) for b in buffers:
  (7)     for method in b:
  (8)     call(method)
  (9) if not selectEvent(event) return
  (10)  output.save()
```

Steps (1) and (2) are performed in the TNM constructor, steps (3) to (5) are done at the first call to beginRun, while steps (6) through (10) are performed in the analyze method. The event loop is under the control of cmsRun. Steps (2) and (4) are the most challenging. In these steps, a key problem is solved: mapping the *name* of a method specified in the configuration file to the method itself.

1.3.1 Some technical details

More key technical details about the implementation of TNM are given in the Appendices. Here we focus on the main features of the implementation. To anchor the discussion, consider the annotated configuration file fragment below, which specifies which methods are to be called for every pat::Muon object.

```
(1)               patMuon =
                  cms.untracked.
                  vstring(
(2)     "patMuon                          cleanLayerMuons                  10",
(3)     "    int    charge()",
        " double    eta()",
        " double    phi()",
        " double    pt()",
        "  float    ecalIso()",
        "   bool    isGlobalMuon()",
(4)     " double    ecalIsoDeposit()->candEnergy()"
        ),
```

Item (1), patMuon, is the name of the *configuration buffer block*, or *config block* for short. A config block is a descriptor that contains the list of methods to be called. The name of a config block must be unique within the configuration file, but is otherwise arbitrary. By default, the name of the config block is the same as the name—the first element in item (2)—of the associated *buffer* object. A buffer is a C++ object, modeled as a C++ template class called Buffer, implemented as an EDM plugin. During the automatic generation of the plugin files (see Section 2.1), the plugin code generator (mkplugins.py) determines whether a buffer is destined to handle a SINGLETON object or a COLLECTION object. By a singleton object we mean an object for which there is at most one per event (for a given getByLabel input tag), while a collection object can contain zero or more objects per event.

In the example above, the patMuon buffer will call the six simple methods, item (3), and the compound method, item (4), for every pat::Muon object, for every event containing such objects. The second element of item (2) is the label to be used in getByLabel (referred to as a category below) for extracting the desired kind of pat::Muon objects from the event, while the third element of item (2) is the maximum number of pat::Muon objects from which data are to be retrieved using the specified methods.

Every buffer contains a sequence of objects that represent the methods to be called by that buffer.

Each method to be called is modeled using the template class `Method`, which delegates the real work to the C++ class `ClassFunction`. Every object of this class contains one or more `FunctionDescriptor` objects that encode what is to be called and how.

A simple method returning a fundamental type has one `FunctionDescriptor`, whereas, a compound method is represented by a sequence of `FunctionDescriptors` arranged in the same order as the components of the compound method, moving from left to right. In the `patMuon` example, the method `int charge()` is modeled by an object of type `Method<pat::Muon>`. That object creates a `ClassFunction` via the constructor `ClassFunction("pat::Muon", "charge()")`, which creates a single `FunctionDescriptor` to represent the method `int charge()`. A single descriptor is sufficient because `int charge()` is a simple method returning a fundamental type. On the other hand, `double ecalIsoDeposit()->candEnergy()` is a compound method requiring two function descriptors, one for `ecalIsoDeposit()`, which returns a pointer to an `IsoDeposit`, and another for `candEnergy()`, which returns a `double`.

The primary task of the `TNM` constructor is to decode the configuration file and make the decoded information available to its `beginRun` method. On the first call to `beginRun`, a buffer is created for every config block in the decoded configuration information. (Buffer creation is deferred until `beginRun` because it is only then that information about High Level Triggers (HLT) is available.) In the `analyze` method, `TNM` loops over every buffer, calling its `fill` method, which in turn loops over and calls the buffer's list of methods using the information stored in the sequence of cached `FunctionDescriptors`. The values returned by the methods are stored in a `TTree` called `Events` whose branches are defined by the buffers. The modeling and invocation of methods is done using the `ROOT 5 Reflex` mechanism[7]. (See Appendix for details.)

`TNM` can handle methods that return objects by *value*, *reference*, *pointer* or EDM *smart pointer*. An object returned by value is a copy of the object internal to the method. An object returned by reference is the object itself, while an object returned by pointer is accessed using the pointer to it. For each return mechanism, `TNM` allocates sufficient memory to store whatever is returned and deallocates the memory when it is no longer needed. For compound methods that return pointers, smart or otherwise, in intermediate steps, `TNM` checks the validity of the pointers before attempting to use them.

Life is simple when a pointer is simple; `TNM` merely checks that the pointer is non-zero. If the pointer is non-zero, it is presumed that all is well and `TNM` proceeds by calling the next component of the compound method using the object pointed to by the returned pointer. For EDM smart pointers life is not so simple because, unfortunately, it turns out not to be sufficient merely to check the value of the pointer. (It is unclear whether this is a feature or a bug.) If the pointer is an EDM smart pointer, `TNM` calls the smart pointer's `isAvailable` method and checks the return value. If the return value is `true`, then `TNM` calls the smart pointer's `isNull` method and checks its return value. If that is `true`, then `TNM` calls the pointer's `get` method, which returns a simple pointer whose value is also checked. `TNM` will abort a call to a compound method if a simple pointer is zero or if either of the calls to `isAvailable` or to `isNull` returns `false`. All information required to call methods efficiently is cached within the `FunctionDescriptors`. The call to methods then reduces to looping over sequences of descriptors and executing the instructions they contain.

2 Getting Started

2.1 Installation

`TheNtupleMaker` must be installed in a standard local release of `CMSSW`. For non-CMS collaborators, the easiest way to use `CMSSW` is to install it within a `docker` container. This can be done by following the instructions at

https://github.com/hbprosper/TheNtupleMaker

[7] This is no longer available in `ROOT` 6.

For concreteness, let's assume we are using the CMS Open Data docker image

```
cmsopendata/cmssw_5_3_32
```

and the name of the docker container we have created is `tnm`. After its creation and exiting from it, the docker container can be restarted using the command

```
docker start -i tnm
```

By default, the `cmsopendata` image sets the initial directory (folder) to be `$HOME/CMSSW_5_3_32/src`. (If that is not the case, move to that directory within the container.) Assuming we are in the `src` directory of the `CMSSW` release, `TNM` can be installed as follows

```
cmsenv
mkdir PhysicsTools
cd PhysicsTools
git clone git://github.com/hbprosper/TheNtupleMaker
cd TheNtupleMaker
scripts/initTNM.py
scram b clean
scram b
```

The build on a fast laptop takes about 10 minutes or less real-time. If this is too slow for you and you have a multi-core machine, you can speed up the build using the job switch `-j` with `scram b`. For example, to run the build using 4 parallel jobs (presumably, one on each core), do

```
scram b -j 4.
```

`TNM` is released with plugins for a few standard helper classes (see Section 3.2 for information on how to create your own helpers.) The remaining plugins are created using the script `initTNM.py`. (See Appendix A for a description of the tasks performed by `initTNM.py`). The `scram b` makes the contents of your python directory (such as `cmsRun` configuration (config) fragments) accessible. The script `mkpackage.py` is located in `PhysicsTools/TheNtupleMaker/scripts`.

2.2 Defining the ntuple

The contents of the ntuples are defined in a configuration file (config file) using a simple self-describing syntax. The config file

```
PhysicsTools/TheNtupleMaker/python/ntuple_cfi.py
```

shows an example of the syntax for specifying the data elements. It is always possible to write the config file by hand, but this requires exact knowledge of class names, class methods and object categories, that is, the labels for `getByLabel`, for the events in the input `ROOT` files, typically, `miniAODs`. `TNM` overcomes this difficulty with a Graphical User Interface (GUI) that

- scans the input file;
- lists all methods that return fundamental types as well as all available object categories;
- provides a way to select the classes, methods, and categories of interest,
- and automatically writes a configuration file fragment in the local `python` directory listing the names of the methods to be called.

The GUI requires a `ROOT` file containing the EDM data objects from which data are to be extracted and written to the ntuple. To run the GUI, do

```
mkntuplecfi.py
```

The first time this Python script is run it will scan through and analyze all the class hierarchies in the `DataFormat` areas of the release before launching the GUI. In subsequent runs, the scanning will not

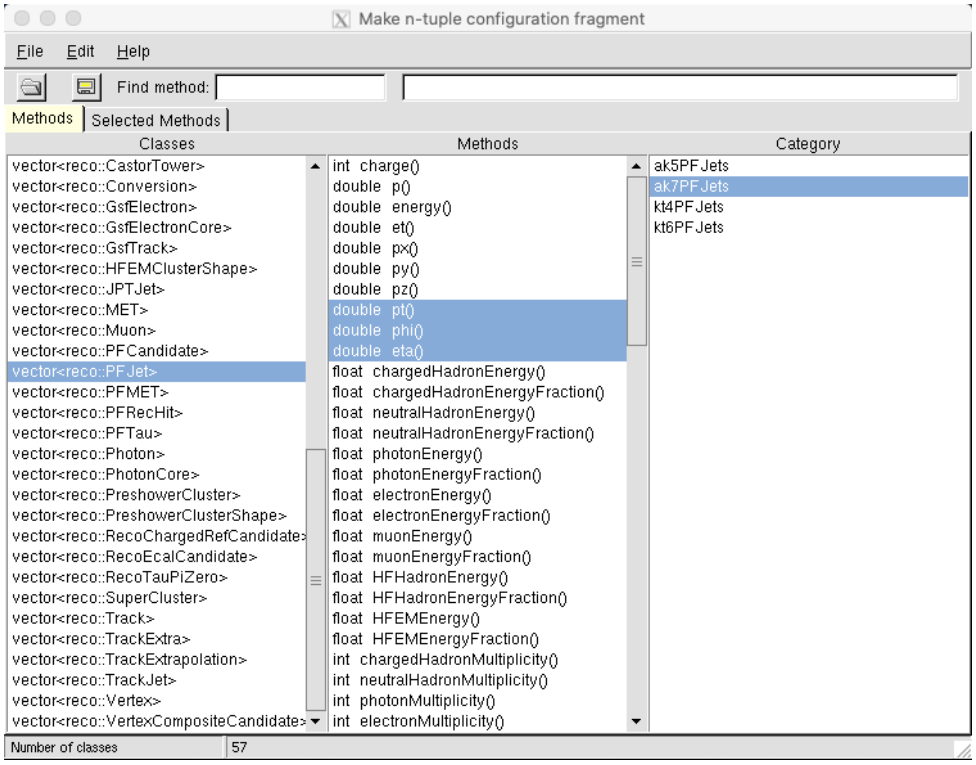

Figure 1: GUI for creating a configuration file that defines the methods to be called by `TNM`. In the figure, the `Methods` tab is selected (and highlighted); the class `vector<reco::PFJet>` has been selected, with methods `pt()`, `phi()`, and `eta()`, and the selected category (that is, the label for `getByLabel`) is `ak7PFJets`. More than one category may be selected.

take place unless one has moved to a different directory, and the GUI will pop up after a few seconds. Once the GUI, shown in Fig. 1, is up and running, do the following:

1. Select `File/Open` from the menu or press the "`Open an EDM file`" button (at the top left of the GUI).
2. From the window that appears, select a `ROOT` file that exemplifies the data set you would like to work with, and press `Open`.
3. The root file will be scanned and all classes identified that match the defined classes of the `CMSSW` release, and which provide methods returning simple types, will be listed in the `Classes` column.
4. Select a class of interest (e.g., `vector<reco::PFJet>`). The selected class will be highlighted. All methods of the selected class that return simple types will appear in the `Methods` column and all labels available for the class will appear in the `Category` column. The GUI lists simple methods, as illustrated in Fig. 1, such as `pt()`. If a class has compound methods, such as

   ```
   int gsfTrack()->numberOfValidHits()
   ```

   they will also be listed. Methods that take arguments with fundamental types are also listed. Moreover, even though the GUI does not list methods such as

   ```
   int track()->hitPattern().numberOfValidMuonHits()
   ```

   `TNM` can handle them.
5. Select one or more categories (that is, the label for `getByLabel`), e.g., `ak7PFJets`. The selected categories will be highlighted.
6. Select methods of interest (e.g., `double pt()`, `double phi()`, `double eta()`). The selected

methods will be highlighted. You can use the "Find method" text entry window to find the methods more easily, but, in the current version of TNM you need to enter the *exact* name of the method and press return[8]. The name of that method will be listed at the top of the list. In the (likely) case that there is more than one method with the same name, pressing return multiple times will loop through the available methods.

7. Repeat steps 4, 5, 6 for all the other classes of interest. (The selected classes will be highlighted in yellow in the Classes column.)
8. When the selection is done, press the Selected Methods tab to check the selection. Only classes for which at least one method and at least one category have been selected will appear here. Click on a class name to see the selected content. One can always cancel a selection by going back to the Methods tab and clicking the names of methods and categories. The highlight will disappear and the item will not be seen among the selected items. A class can be removed from the selected classes by clicking on the class to be removed, thereby listing its contents, and clicking on the class again to remove it from the selected classes.
9. When the selection is complete, choose File/Save from the menu or press the Save button (the second button in the toolbar) to save the configuration file fragment. The default name for the configuration file fragment is ntuple_cfi.py and, by default, it is saved to the local python directory. If you have chosen to save the configuration file in a place other than the python directory be sure to copy the config file to that directory after you exit the GUI.
10. Exit the GUI by following the usual procedure for your operating system, or go to File and choose Exit.

The configuration file

```
python/ntuple_cfi.py
```

will include all the selected items. You are free to make any changes, by hand, conforming with the format, such as adding methods not listed by the GUI. A complete (annotated) example of ntuple_cfi.py is given in Appendix B.

In the current version of TNM, a special case arises that requires intervention by hand, namely, when methods require one or more arguments, e.g., float bDiscriminator(std::string). When such a method is selected, the configuration file created with the GUI will contain the entry

```
float bDiscriminator(std::string)
```

One needs to replace the argument type, here std::string, with its actual value: characters written within double quotes. Furthermore, you should define an alias at the end to distinguish the name of the leaf in the ntuple from other leaves that could possibly be made from the same method. The corrected version of the above entry becomes

```
float bDiscriminator("simpleSecondaryVertexBJetTags") simpleSecVertexBJetTags
```

In general, the alias is optional because a default name for the ntuple variable is constructed automatically from the method name. In this example, however, we use an alias because the default name may not be suitable.

The GUI is merely an aid to get you started. It lists a large number of the methods that can be accessed by TNM. However, there are many methods accessible by TNM that are currently not handled by the GUI. These are typically compound methods, such as int pdf()->x.first in GenEventInfoProduct. If a compound method comprises a sequence of methods and/or datamembers, and if the arguments of the methods are simple types, it is possible that TNM can access that method. So be experimental!

[8] In the ROOT 6 version of TNM under development, the search will be more sophisticated.

2.2.1 *Special treatment of the triggers*

TNM provides access to the values and prescales for triggers using `int value(trigger-name)` and `int prescale(trigger-name)`. The values are 0 if the trigger did not fire, 1 if it did, or negative if the trigger does not exist. When trigger information is available, the trigger menus are read via `HLTConfigProvider`. In order to be effective triggers must, and do, adapt to changing run conditions. By convention, every time an existing trigger changes it is given a new version number, which appears at the end of the trigger name. Each version of a trigger can be listed explicitly in the configuration file as in the following example.

```
              edmTriggerResultsHelper =
              cms.untracked.
              vstring(
"edmTriggerResultsHelper          TriggerResults::HLT               1",
#-------------------------------------------------------------------
'   int   value("HLT_BeamHalo_v5")',
'   int   value("HLT_BeamHalo_v6")',
'   int   prescale("HLT_BeamHalo_v5")',
'   int   prescale("HLT_BeamHalo_v6")',
'   int   value("HLT_Jet240_CentralJet30_BTagIP_v2")',
'   int   value("HLT_Jet240_CentralJet30_BTagIP_v3")',
'   int   value("HLT_Jet270_CentralJet30_BTagIP_v2")',
'   int   value("HLT_Jet270_CentralJet30_BTagIP_v3")'
'   int   prescale("HLT_Jet240_CentralJet30_BTagIP_v2")',
'   int   prescale("HLT_Jet240_CentralJet30_BTagIP_v3")',
'   int   prescale("HLT_Jet270_CentralJet30_BTagIP_v2")',
'   int   prescale("HLT_Jet270_CentralJet30_BTagIP_v3")'
)
```

However, TNM supports a range and wildcard syntax that can be used to shorten what could otherwise be a long, and tedious to write, list of trigger names. The above example can be shortened as follows.

```
              edmTriggerResultsHelper =
              cms.untracked.
              vstring(
"edmTriggerResultsHelper          TriggerResults::HLT               1",
#-------------------------------------------------------------------
'   int   value("HLT_BeamHalo_v5...6")',
'   int   prescale("HLT_BeamHalo_v5...6")',
'   int   value("HLT_Jet2*_CentralJet30_BTagIP_v*")',
'   int   prescale("HLT_Jet2*_CentralJet30_BTagIP_v*")
)
```

A range is specified with an ellipsis "..."; note also the use of the wildcard character "*". In fact, if you feel particularly adventurous, you can use a regular expression so long as it is not overly complicated.

2.3 Making the ntuple

The next step after specifying the contents of the ntuple is to run `TheNtupleMaker` over `ROOT` files in EDM format. Given the configuration file fragment `python/ntuple_cfi.py`, the next steps are:

1. Go to the directory containing the file `TheNtupleMaker_cfg.py`.
2. Open `TheNtupleMaker_cfg.py` with your favorite editor and modify the `PoolSource`, specifying the full name of your input EDM file (or files).
3. Then do

   ```
   cmsRun TheNtupleMaker_cfg.py
   ```

   to run `TheNtupleMaker` and create the output ntuple.

There will be two outcomes of this run:

– A file called `ntuple.root` containing the ntuple. The name `ntuple.root` is defined in

```
python/ntuple_cfi.py
```

which contains the declaration of `TheNtupleMaker` (an `EDAnalyzer`) one of whose attributes is

```
ntupleName = cms.untracked.string("ntuple.root")
```

and whose value can be changed.

– A directory called `analyzer`, again, by default that includes an automatically created program skeleton that can be used to read and analyze the contents of the ntuple. The name `analyzer` is defined in `ntuple_cfi.py` as follows

```
analyzerName = cms.untracked.string("analyzer.cc")
```

and can be changed.

Note: If you do not want the analyzer to be created automatically (and therefore risk overwriting an existing analyzer of the same name) you should comment out the above line in the config file fragment `ntuple_cfi.py` (in your `python` directory) and create your analyzer explicitly using

```
mkanalyzer.py <analyzer-name>
```

after `TNM` has been run. The script `mkanalyzer.py` reads the file `variables.txt`, which is written by `TNM`, and constructs the analyzer using the information contained therein. The file `variables.txt` can also be created independently of `TNM` using the command

```
mkvariables.py <ntuple-file-name> <tree-name-1> [tree-name-2 ...]
```

For example,

```
mkvariables.py boris.root karloff
```

will create the file `variables.txt` from the tree `karloff` in the ntuple file `boris.root`, which can then be used to create an analyzer using `mkanalyzer.py`. The combination of `mkvariables.py` and `mkanalyzer.py` should work on any simple flat ntuple, not only those created by `TNM`.

2.4 Analyzing the ntuple

The analyzer

– reads the contents of the ntuple and
– maps the contents (that is, the branches) to single variables or to `STL` vector variables with automatically generated names.

If you don't like the automatically generated names, you can edit the file `variables.txt`, which **note** is re-created every time `TNM` is run, and replace all the variable names that annoy you with ones that don't. If you don't want your edits to be overwritten remember to backup your file to a name other than `variables.txt`.

Each line of the file `variables.txt` has 4 fields:

```
<type>/<leaf-name>/<variable-name>/<max-variable-count>
```

The first is the `type` of the variable, the second is the name of the variable in the ntuple (the leaf name), the third is the name to be assigned to the variable, and the last field is the maximum item count for the variable[9]. Feel free to change the variable name fields to names you like. Then (re)create your analyzer using

```
mkanalyzer.py analyzer
```

[9] An asterisk at the end of an entry in `variables.txt` indicates that the variable is a leaf counter.

which creates a directory called `analyzer`, which is where you will analyze the ntuple.

The `TheNtupleMaker` analyzer mechanism is designed to be independent of `CMSSW`, with a dependency only on `ROOT` 5 and a standard Unix-like environment. Two versions of the analyzer are available: one in C++ and one in Python. Below are the instructions for using the C++ version of the analyzer (which, we assume is called `analyzer`).

1. `cd analyzer`
2. The header file `analyzer.h` has all necessary includes and is the place where the variables are defined. You do not need to, and should not, modify this file. You can use the header as a reference for the variable names available to you. The header also defines various methods that are useful for skimming events, adding weights, customizing the definition of command line arguments, etc. Please see the relevant methods for details.
3. The source code `analyzer.cc` is a traditional event-oriented system that loops over the events read from the ntuple. This is the code to be modified. You can book histograms and write analysis code at the places noted by inline comments. Feel free to add branches, trees, etc. to the output root file.
4. Compile using `make`, which creates the executable `analyzer`.
5. The executable `analyzer` is run as follows

   ```
   ./analyzer [name of file listing the names of ntuple files to be read]
   ```

   By default the name of the file containing the list of ntuple files to be read is `filelist.txt` and may be omitted. The text file `filelist.txt` lists the names of the ntuple files with one name per line, e.g,

   ```
   ../ntuple_1.root
   ../ntuple_2.root
   ../ntuple_3.root
   ```

   The result of the run is a `ROOT` file called `analyzer_histograms.root`, which contains the histograms you created, including any directory structure you may have imposed upon them.

2.4.1 Alternative analysis frameworks

You are not obliged to use the `TNM` analyzer, which is a traditional event-oriented framework, to analyze the contents of such ntuples. The current trend is to view particle physics data as simply huge multi-dimensional tables rather than as a sequence of events. Several powerful state-of-the-art tools are available with extremely efficient implementations of this point of view, such as Jim Pivarski's `Python uproot` module, which is available from `https://pypi.org/project/uproot/` and `ROOT`'s `RDataFrame` declarative analysis framework.

3 Advanced Features

The details given in the previous sections are sufficient for many analyses. However, there are times when some intervention mechanisms are needed to support the automatic ntuple generation. This section describes these features of `TNM`.

3.1 User-defined variables

The purpose of `TNM` is to automate the creation of ntuples from the thousands of methods and data members in the `DataFormats` area of `CMSSW` that return fundamental types. Sadly, however, this is sometimes not enough! Therefore, `TNM` also provides a *helper mechanism* to handle composite variables (e.g., `deltaphi(jet pT, MET)`). In addition, there is also a mechanism for calling a macro from `TNM` that can be used to add user-defined variables to the ntuple. We describe both of these mechanisms below. Ideally, even this would be automated in a cutting edge system.

3.2 The helper mechanism

A helper is a class that extends the interface of another class. A helper for `pat::Muon`, for example, inherits all the valid methods of `pat::Muon`, as well as the additional methods provided by the helper. Moreover, a helper behaves very much like any other class. It is associated with a buffer (in this case, a buffer of type `UserBuffer`) and it is called in exactly the same way as far as `TNM` is concerned. However, a `UserBuffer`, in contrast to a `Buffer`, provides several places in which a helper can interact with the current `edm::Event` as well as with the helped object.

Typically, helpers are used to add methods that may need access to, and the manipulation of, one or more objects in order to make available numbers that are not directly available via existing methods. For example, the access to trigger information in `CMSSW` is not as easy as it ought to be so a helper for the `TriggerResults` object is needed. Its purpose is to provide two obviously useful methods: `int value(trigger-name)`, which returns the trigger value (0 if the trigger did not fire, 1 if it did or negative if the trigger does not exist) given the trigger name and `int prescale(trigger-name)`, which returns the pre-scale value associated with the specified trigger. A trigger with a pre-scale of 10 would record only 1/10 of the triggered events.

You can produce a helper for an object by running the `mkhelper.py` script as follows:

```
mkhelper.py [options] <CMSSW class> s|c [postfix, default=Helper]
```

where the `CMSSW` class should be given as `namespace::Class`. The `s|c` denote whether the object is a singleton (i.e., there is at most one object per event, like `edm::Event`), or a collection (i.e. there are zero or more objects in the event, like `vector<reco::PFJet>`. One can also modify the postfix of the helper name, which is by default `Helper`. For example, to make a helper for a `pat::Jet` class execute the command

```
mkhelper.py pat::Jet c
```

while for a singleton like `edm::Event`, with a postfix `Extended`, you execute

```
mkhelper.py edm::Event  s  Extended
```

Running the script creates the helper class in the `src` and `interface` directories (the examples above will create `patJetHelper` and `edmEventExtended`). After making the necessary changes, compile the helper with

```
scram b
```

The `helper.h` file in the `interface` directory gives a comprehensive description of the accessible variables and methods. The following variables are automatically defined and available to all methods:

```
blockname           name of config. buffer object (config block)
buffername          name of buffer in config block
labelname           name of label in config block (for getByLabel)
parameter           parameter (as key, value pairs)
                    accessed as in the following example:

                    string param = parameter("label");

0. hltconfig        pointer to HLTConfigProvider
                         Note: this will be zero if HLTConfigProvider
                         has not been properly initialized

1. config           pointer to global ParameterSet object
2. event            pointer to the current event
3. object           pointer to the current helped object
4. oindex           index of current helped object
5. index            index of item(s) returned by helper.
                         Note 1: an item is associated with all
```

```
                        helper methods (think of it as an
                        extension of the helped object)

                        Note 2: index may differ from oindex if,
                        for a given helped object, the count
                        variable (see below) differs from 1.
6. count            number of items per helped object (default=1)
                        Note:
                        count = 0  ==> current helped object is
                                       to be skipped

                        count = 1  ==> current helped object is
                                       to be kept

                        count > 1  ==> current helped object is
                                       associated with "count"
                                       items, where each item
                                       is associated with all the
                                       helper methods

variables 0-6 are initialized by TheNtupleMaker.
variables 0-5 should not be changed.
variable  6 can be changed by the helper to control whether a
          helped object should be kept or generates more items
```

The helper class contains the following methods:

- All methods of the original class (e.g., all methods of the `pat::Jet` class)
- `virtual void analyzeEvent()`: called once per event. This can be used, for example, for retrieving collections
- `virtual void analyzeObject()`: called once per object (e.g., once per every `pat::Jet`). This can be used for adding new variables, skimming based on object selection, etc.

In order to add a new variable `type x`, for example, `double dphijetmet` to each `pat::Jet`, do the following:

1. In `helper.h`, declare the access method as
   ```
   double dphijetmet() const;
   ```
2. Calculate `dphijetmet` in `analyzeObject()` or `analyzeEvent()`, whichever is the more appropriate, and cache the result.
3. Make the method `double dphijetmet()` return `dphijetmet`:
   ```
   double JetHelper::dphijetmet()  const
   {
     return  dphijetmet_;
   }
   ```
   where `dphijetmet_` is the variable to which the value of $\delta\phi(jet, MET)$ is assigned.

To add new variables to an event, for example H_T, you can create a helper for the `edm::Event` object as described above. However, there already exists a default `edmEventHelper` in the `TNM` package, which is designed to retrieve `edm::Event` methods such as `run()`, `event()`, `luminosityBlock()`, etc., therefore, a new `edm::Event` helper should have a different name e.g., `edmEventExtended`. The helper mechanism can also be used for skimming objects. This will be described in Section 3.4.

3.3 The macro mechanism

`TNM` provides a mechanism for calling a compiled `ROOT` macro *after* all methods have been called but *before* the retrieved data have been committed to the tree `Events`. This provides the macro with the

opportunity to direct TNM to skim events as well as objects. A macro has access to all the variables of the ntuple's tree, to which it can also add branches. (See Appendix C for some technical details.)

A macro template is created from the `variables.txt` file in much the same way as an analyzer. For example,

```
mkmacro.py mymacro
```

will create the files

```
mymacro.cc
mymacro.h
mymacro.mk
```

The source code `mymacro.cc` contains a commented example of how to add a variable, for example "H_T", to the ntuple. The header, which generally one should not edit, lists all the ntuple variables that are available automatically. After editing `mymacro.cc` it must be compiled and linked using

```
make -f mymacro.mk
```

to create the shared library `libmymacro.so` before it can used by TNM. You tell TNM about your macro using the `macroName` variable of the `EDAnalyzer` as shown below.

```
cms.EDAnalyzer("TheNtupleMaker",
               ntupleName = cms.untracked.string("ntuple.root"),
               macroName  = cms.untracked.string("mymacro.cc"),
                    :          :
```

The new variables will be added to the ntuples' `Events` tree.

To add HT, do the following:

1. In `struct mymacroInternal`, declare HT
2. In `beginJob()`, add the HT branch to the tree:
   ```
   tree->Branch("HT", &local->HT, "HT/F")
   ```
3. In `analyze()`, calculate the value for HT and assign it to `local->HT`.

Note 1: Remember to compile and link your macro before you call `cmsRun`.

Note 2: The variable type must match the type given to the `tree.Branch(...)` method. For example, if your variable is called HT and its type is a `float`, then its type specifier in the `tree.Branch(...)` method must be "HT/F"; likewise if it is a `double` its type specifier must be "HT/D" and so on. If you find weird values for your variables, it may be because you have a type mismatch.

The macro mechanism can also be used for skimming. This will be described in Section 3.4.

3.4 Skimming

The macro mechanism can be used for skimming events and skimming objects. The helper mechanism can be used for skimming objects.

3.4.1 Skimming events

By default, the macro returns `true` (see the end of `mymacro.cc`). This means that the event is written into the ntuple. If you would like to skip an event for a certain condition, you should make the macro return `false` given that condition.

3.4.2 Skimming objects

Using the macro mechanism: The source code `mymacro.cc` contains a commented example of how to skim an object.

1. In `beginJob()`, define the objects that are to be selected in `analyze()`, e.g.: select("jet"); The available object names are listed in `mymacro.h`.
2. In `analyze()`, loop over the objects and select the objects that satisfy the necessary condition, e.g.:

```
for(size_t i=0; i < jet_pt.size(); ++i)
{
  if ( !(jet_pt[i] > 100) ) continue;
  select("jet", i);
}
```

3.4.2.1 Using the helper mechanism

The variable `count` determines whether an object is kept or skipped. By default `count = 1` and the object is kept. To skip an object for a given condition, set `count = 0` for that condition in the `analyzeObject()` method.

3.5 Controlling the `CMSSW MessageLogger`

`TNM` uses the `CMSSW MessageLogger`. Therefore, it is possible to modify the frequency of messages by placing `MessageLogger` commands in the `TheNtupleMaker_cfg.py` configuration file. By default, all messages from the `MessageLogger` are reported. However, consider the example below.

```
(1) process.load("FWCore.MessageService.MessageLogger_cfi")
(2) process.MessageLogger.destinations = cms.untracked.vstring("cerr")
(3) process.MessageLogger.cerr.FwkReport.reportEvery = 10
(4) process.MessageLogger.cerr.default.limit = 5
```

(1) This line, which is already present in `TheNtupleMaker_cfg.py`, loads the default configuration of the `MessageLogger`.

(2) Tell the `MessageLogger` that you intend to modify the behavior of messages destined for the output stream `cerr`.

(3) Tell the `MessageLogger` to report `CMSSW Framework` messages once every 10 events.

(4) Tell the `MessageLogger` to limit the number of messages to 5 and, thereafter, exponentially reduce the number of messages.

3.6 `TNM` and provenance

It particularly important to have access to the *provenance* information of a given sample, that is, the origin and history of the sample and how the contents of the sample were constructed. `TNM` is a self-documenting system and is designed to store necessary provenance information in the ntuples it creates. The following provenance information can be accessed from the ntuples and more can be added if needed.

– The variable names in the `Event` tree are constructed, by default, as

```
<objectname>_<category>_<methodname(s)>
```

where category is the label given to the `getByLabel` function. The name of the variable reflects its EDM origin.

– The resulting ntuples also contain a `Provenance` tree, which records the following information

```
cmssw_version
date
hostname
username
cfg
```

Here, `cfg` is the `ntuple_cfi.py` that was used for creating the ntuple, and as explained above, this file contains the EDM information of all variables stored in the ntuple, i.e., their classes, categories (`getByLabel` labels) and methods.

4 Summary

This document presents `TheNtupleMaker`, a standard system for automating the creation of ntuples for the analysis of CMS data in EDM format. The version of `TNM` described in this document works with `ROOT` 5 and relies heavily on the `ROOT` C++ reflex mechanism. Since `ROOT` 6 no longer supports this mechanism a new version of `TNM` is under development, which use the just in time compiler introduced with `ROOT` 6. `TheNtupleMaker`

– makes flat ntuples from all CMS EDM formats including `RECO`, `AOD`, `PAT`;
– allows for the addition of user-defined variables to the ntuples;
– allows for object and event skimming;
– automatically generates an analyzer for reading and analyzing the generated ntuples
– is self-documenting and stores relevant provenance information.

In a time that is so exciting for LHC physics, it is beneficial to create tools that flatten the learning curve so that those with modest computing skills can nevertheless make intellectual contributions to LHC physics. `TNM` was designed with this aim in mind. We are convinced that today it ought to be possible for a physicist to state what she wishes to do in an intuitively natural way and delegate to the computer the chore of routine coding. While we recognize that not everything can be automated, we believe that a great deal of routine data processing can and should be automated. `TheNtupleMaker`, the analysis description language (`ADL`) project, and similar efforts are all an attempt to move towards analysis platforms that free the analyst from unnecessary complexity while she focuses on what truly matters: thinking creatively about the science forwards.

Acknowledgements

We thank the many people who, like us, believe that physics should be about physics and, therefore, the tools we use for doing physics should not be overly complicated beyond absolute necessity. These colleagues helped test and improve `TNM`. Many thanks to Jeff Haas for his support since the very beginning; to Joe Bochenek and Lukas Vanelderen for their extensive real-world usage of `TNM` and giving valuable feedback; to our colleagues Don Teo, Josh Thompson, Ben Kreis, Harold Nguyen and Sudan Paramesvaran; to Nadja Strobbe and Piet Verwilligen; to Supriya Jain, and to colleagues Maurizio Pierini, Francesco Santanastasio and Maxime Gouzevitch for adopting `TNM` in their analyses and making many useful suggestions. We also thank Maria Spiropulu for her strong support.

Appendix

A Initializing TNM

The script scripts/initTNM.py performs the following tasks.

1. Make the Python scripts in the scripts directory executable.
2. Create the directory

   ```
   $CMSSW_BASE/python/PhysicsTools
   ```

   if it does not exist.
3. Create a soft link

   ```
   $CMSSW_BASE/python/PhysicsTools/TheNtupleMaker
   ```

   that points to
4. Run mkclasslist.py to create the file plugins/classlist.txt listing the classes for which buffer plugins are to be made. This script uses python/classmap.py to determine which package src directories to scan for classes_def.xml files. For each XML file found, the class names are extracted from the entries containing the keyword edm::Wrapper. A class that ultimately maps to an STL vector, such as vector<pat::Muon>, is considered a collection, otherwise it is taken to be a singleton. Complicated beasts such as edm::AssociationMap are skipped, as are classes which are listed in the file plugins/exclusionlist.txt. Such classes must be handled using helpers (see below). If mkclasslist.py fails to include a class that you want (and the class is not a beast), you should list it (one class name per line) in the file plugins/inclusionlist.txt. If mkclasslist.py, via mkclassmap.py, can find its header, then the classes listed in plugins/inclusionlist.txt will be added to plugins/classlist.txt.
5. Finally, for each class listed in plugins/classlist.txt, the script mkplugins.py creates an entry for that class in one of several plugin files that are created in the plugins directory.

B TNM configuration file

The configuration file fragment ntuple_cfi.py, which resides in the python directory of a package, contains the instructions to control TNM. In particular, ntuple_cfi.py specifies what methods are to be called by TNM to create an ntuple. As described in Section 2.1, a good first draft of ntuple_cfi.py can be created using the GUI.
The numbers below correspond to those in the annotated example that follows.

(1) Specify using the standard CMS configuration file syntax

```
some-object = cms.EDAnalyzer(module-name,
      attribute-1 = cms....,
      attribute-2 = cms....,
        ::
    )
```

that an EDAnalyzer, namely TNM, is to be dynamically loaded into and run by cmsRun. ntupleName (2), analyzerName (3), and buffers (4a) - (4b) are the main attributes of TNM.

(2) This specifies the name of the ntuple to be created.

(3) This specifies the name of the ntuple analyzer to be created.

(4a) - (4b) This lists the config blocks, for example edmEvent, that TNM should expect to find and decode. The name of a config block must be unique within the configuration file, but is otherwise arbitrary. However, by default the config block has the same name as the buffer, unless

that would violate the requirement that their names be unique. If more than one config block needs to be specified, each referring to the same buffer but with a different `getByLabel` tag, then uniqueness is achieved, by default, by the expedient of adding a number to the name of the config block. In this example, `TNM` will expect to see config blocks for the buffers `edmEvent`, `HcalNoiseSummary`, `recoCaloJet` and `edmTriggerResultsHelper`.

(5a) - (5b) This is the config block for the buffer `edmEvent`. Notice that this config block is special in that it does not have a `getByLabel` tag since this is not needed for an `edm::Event`.

(6a) - (6b) This is an example of a config block for one of the standard helpers. By design, the config block syntax for buffers and for buffers that are also helpers is the same. However, config blocks for helpers can have parameters. For each parameter, the syntax is

```
' param parameter-name = parameter-value',
```

Note: param is a reserved keyword that tells `TNM` that what follows is a key-value pair. The parameters can be accessed in a helper using

```
std::string paramstr = parameter(paramname);
```

where `paramname` is the key, that is, the name of the parameter, and `paramstr` is its value returned as an STL string, which can be decoded as the helper writer sees fit.

It was a deliberate design choice to keep the syntax of the configuration file as clean and simple as possible so that the file would be easy to read. Basically, everything is specified using strings. Experience suggests that the most common syntax error that can occur is failure to delimit one or more strings with commas. Naturally, this provokes a runtime decoding error that causes `TNM` to give up in disgust.

Example `ntuple_cfi.py` :

```
#-----------------------------------------------------------------------------
# Created: Tue Aug 24 22:06:57 2010 by mkntuplecfi.py
#-----------------------------------------------------------------------------
import FWCore.ParameterSet.Config as cms
demo = cms.EDAnalyzer("TheNtupleMaker",                                  (1)
               ntupleName = cms.untracked.string("ntuple_reco.root"), (2)
               analyzerName = cms.untracked.string("analyzer_reco.cc"),  (3)

               buffers =              (4a)
               cms.untracked.
               vstring(
    'edmEvent',
    'HcalNoiseSummary',
    'recoCaloJet',
    'edmTriggerResultsHelper'
    ),                                (4b)

               edmEvent =             (5a)
               cms.untracked.
               vstring(
    'edmEvent',
    #-----------------------------------------------------------------------
    '   int   isRealData()',
    '   int   id().run()',
    '   int   id().event()',
    '   int   luminosityBlock()',
    '   int   bunchCrossing()',
    '   unsigned int time().unixTime()',
    '   unsigned int time().nanosecondOffset()'
    ),                                (5b)
```

```
            HcalNoiseSummary =
            cms.untracked.
            vstring(
'HcalNoiseSummary                hcalnoise                          1',
#-------------------------------------------------------------------
'   bool   passHighLevelNoiseFilter()',
'   bool   passLooseNoiseFilter()'
),
                recoCaloJet =
            cms.untracked.
            vstring(
'recoCaloJet                     ak5CaloJets                      500',
#-------------------------------------------------------------------
' double   energy()',
' double   eta()',
'  float   etaetaMoment()',
' double   phi()',
'  float   phiphiMoment()',
' double   pt()',
'  float   emEnergyFraction()',
'    int   n90()'
),
            edmTriggerResultsHelper =                    (6a)
            cms.untracked.
            vstring(
"edmTriggerResultsHelper          TriggerResults::HLT               1",
#-------------------------------------------------------------------
'   int   value("HLT_Jet15U")   Jet15U',
'   int   value("HLT_Jet30U")   Jet30U',
'   int   value("HLT_Jet50U")   Jet50U',
'   int   value("HLT_L1Jet6U")  L1Jet6U'
)                                                        (6b)
            )
```

C Technical Descriptions

Health Warning: FOR GEEK EYES ONLY

This Appendix provides the most important technical details of the key software mechanisms that underpin TNM.

C.1 The Reflex mechanism

Reflex is a set of C++ classes, distributed with ROOT 5 (see $ROOTSYS/include/Reflex), that provide a high-level interface to class dictionaries as well as a mechanism to invoke class methods. A class dictionary, which can be created and linked into a shared library using scram, provides detailed information about a class that can be accessed at runtime. In CMSSW, dictionary creation is controlled with the files classes.h and classes_def.xml, which are located in the src directory of a CMSSW package.
The key tools in Reflex are

(a) the function ROOT::Reflex::Type::ByName(classname), which given the name of a class returns an object of type ROOT::Reflex::Type that represents the class,

(b) ROOT::Reflex::Member that represents a class data member or method and

(c) ROOT::Reflex::Object that represents an object, for example, an object (which could be a fundamental type) returned by a method.

In order to provide a more convenient interface for use by TNM, the Reflex mechanism is encapsulated in the (TNM) class ClassFunction, whose usage is illustrated in the following example.

```
ClassFunction f("pat::Muon", "gsfTrack()->phi()");
  :  :
double phi = f(muon),
```

which is equivalent to

```
double phi = muon.gsfTrack()->phi().
```

The next section describes the method invocation mechanism.

C.2 Class analysis and method invocation

CMS classes tend to have deep inheritance hierarchies, which complicates the method invocation mechanism. Here, for example, is the inheritance hierarchy of the class `pat::Electron`:

```
pat::Lepton<reco::GsfElectron>
  pat::PATObject<reco::GsfElectron>
    reco::GsfElectron
      reco::RecoCandidate
        reco::LeafCandidate
          reco::Candidate
```

In order to call a method using the `Reflex` mechanism, it is necessary to determine to which class the method belongs. This requires a sequential search of the inheritance hierarchy, starting with the derived class, until the name and arguments (that is the signature) of the method to be called matches a method in one of the classes in the hierarchy. For example, a search for the method `gsfTrack()` will achieve a match in the derived class `pat::Electron`, whereas a search for the method `pt()` will achieve a match in the base class `reco::LeafCandidate`, the fifth base class encountered in the inheritance hierarchy. For overloaded methods, the class `ClassFunction` follows the C++ inheritance rule: the first method that matches is assumed to be the method to be called. The matching is done with regular expressions.

C.3 `ClassFunction`

This class is the real workhorse of the entire system. It relies heavily on `Reflex`. Should `Reflex` fail, so too would `TNM`! Perhaps the simplest way to describe how `ClassFunction` works is with pseudocode. Below is a “Pythonesque" rendition of `ClassFunction`.

```
ClassFunction.constructor(classname, expression):
    #--------------------------------------------------------------------
    # Split method into its parts. Surely, even for CMS, a maximum
    # indirection depth of 20 is sufficient!
    #--------------------------------------------------------------------
    maxDepth = 20
    # This flag has to be true at the end of this routine, otherwise
    # something is wrong.
    done = False
    #--------------------------------------------------------------------
    # Note:
    # classname  - type name of parent object
    # expression - method/datamember of parent object
    # rname      - type name of returned object or data member
    # fd_        - a vector of function descriptors
    # depth      - the depth of a compound method
    #--------------------------------------------------------------------
    for depth in [0...maxDepth-1]:
        # Allocate a function descriptor for each component of a
        # compound method
        fd_.push_back( FunctionDescriptor() )
        fd = fd_[depth]
        # NB: get a reference NOT a copy!
```

```
        fd.classname    = classname
        fd.expression   = expression
        fd.otype        = Type::ByName(fd.classname)
        # Model parent class
        if fd.otype.Name() == "":
            haveAtantrum("cannot get class info")

        # initialize function descriptor
        fd.datamember   = False
        fd.simple       = False
        fd.byvalue      = False
        fd.pointer      = False
        fd.reference    = False
        fd.smartpointer= False
        fd.isAvailable  = False
        fd.isNull       = False

        # If method is compound, split it in two and set "expression"
        # to the first part of the compound method and "expr2" to the
        # remainder

        delim = ""  # delimeter between expression and expr2
        expr2 = ""
        if isCompoundMethod(expression, delim):
            bisplit(expression, fd.expression, expr2, delim)

        # Determine whether this is a function or data member
        dregex = boost.regex("^[a-zA-Z_]+[a-zA-Z0-9_:]*[(]")
        dmatch = boost.smatch()
        fd.datamember = not boost.regex_search(fd.expression,
                                               dmatch, dregex)

        if fd.datamember:
            #-------------------------------------------------------------
            # This seems to be a data member
            #-------------------------------------------------------------
            # Get a model of it
            fd.method = getDataMember(fd.classname, fd.expression)

            # Fall on sword if we did not find a valid data member
            if not memberValid(fd.method):
                haveAtantrum("can't decode data member")

            # We have a valid data member. Get type allowing for
            # the possibility that values can be returned
            # by value, pointer or reference.
            # 1. by value     - a copy of the object is returned
            # 2. by pointer   - a variable (the pointer) containing the
            #                   address of the object is returned
            # 3. by reference - the object itself is returned
            fd.rtype    = fd.method.TypeOf()
            # Model type of data member
            fd.simple   = fd.rtype.IsFundamental()
            fd.pointer  = fd.rtype.IsPointer()
            fd.reference = fd.rtype.IsReference()
            fd.byvalue = not (fd.pointer or fd.reference)

            # Get type name of data member
            # Note: for data members, the rtype variable
            # isn't the final type in the sense that it}
            # could still include the "*" or "&" appended to the
            # type name. However, for methods rtype is the final
```

```
            # type.
            fd.rname = fd.rtype.Name(SCOPED+FINAL)

            if fd.pointer or fd.reference:
                # remove "*" or "&" at end of name
                fd.rname = fd.rname[:-1]

            # Fall on sword if we cannot get data member type name
            if fd.rname == "":
                haveAtantrum("can't get type for data member")

        else:
            #-------------------------------------------------------------
            # This seems to be a method
            #-------------------------------------------------------------
            # Decode method and return a Reflex model of it in fd.method
            decodeMethod(fd)

            # Fall on sword if we did not find a valid method
            if not rfx::memberValid(fd.method):
                 haveAtantrum("can't decode method")

            # We have a valid method so get a model of its return type
            # Note: again, allow for the possibility that the value can be
            # returned by value, pointer or reference.
            fd.rtype   = fd.method.TypeOf().ReturnType().FinalType()
            fd.simple  = fd.rtype.IsFundamental()
            fd.pointer = fd.rtype.IsPointer()
            fd.reference = fd.rtype.IsReference()
            fd.byvalue = not (fd.pointer or fd.reference)

            # Further decode return type (rtype)
            if fd.pointer: fd.rtype = fd.rtype.ToType()

            # Get type fully scoped name of returned object
            fd.rname = fd.rtype.Name(SCOPED+FINAL)
            if fd.rname == "":
                haveAtantrum("can't get return type for method")

            # This could be an isAvailable method of an EDM so-called smart
 pointer
            aregex = boost.regex("^isAvailable[(]")
            amatch = boost.smatch
            fd.isAvailable = boost.regex_search(fd.expression, amatch,
aregex)

            # Maybe it is an isNull method
            nregex = boost.regex("^isNull[(]")
            nmatch = boost.smatch nmatch
            fd.isNull = boost.regex_search(fd.expression, nmatch, nregex)

        #-----------------------------------------------------------------
        # We have a valid method or data member.
        #-----------------------------------------------------------------
        # The return type or data member could be an EDM smart pointer
        if not fd.simple:
            m = getisNull(fd.method)
            fd.smartpointer = memberValid(m)

        if fd.smartpointer:
            # The data member or the return type is a smart pointer, so
            # insert a call to isAvailable()
```

```
            expr2 = "isAvailable()" + delim + expr2

        elif fd.isAvailable:
            # This is an isAvailable method, so insert a call to isNull
            # The same class name as current smart pointer
            fd.rname = fd.classname
            expr2 = "isNull()" + delim + expr2

        elif fd.isNull:
            # This is an isNull method, so insert a call to get
            # same classname as current smart pointer
            fd.rname = fd.classname
            expr2 = "get()" + delim + expr2

        # Memory is needed by Reflex to store the return values from
        # functions.We need to reserve the right amount of space for
        # each object returned, which could of course be a fundamental
        # (that is, simple) type. We free all reserved memory in
        # ClassFunction's destructor.
        fd.robject = Object(fd.rtype, fd.rtype.Allocate())

        # set return type code
        fd.rcode = Tools.FundamentalType(fd.rtype)

        # ================================================================
        # THIS IS WHERE WE BREAK OUT OF THE METHOD DECONSTRUCTION LOOP
        # If the return type is simple, then we need to break out of this
        # loop because the analysis of the method is complete. However, if
        # the method is either isAvailable or isNull we must continue.
        # ================================================================
        if fd.simple:
            if not fd.isAvailable:
                if not fd.isNull:
                    # ------------------------------------------------------
                    # ClassFunction should always arrive here!
                    # ------------------------------------------------------
                    done = True

        # The return type is not simple or the method is either isAvailable
        # or isNull. We therefore, need to continue: the 2nd part of the
        # compound method becomes the expression on the next round and the
        # return type becomes the next classname.

        expression = expr2
        classname  = fd.rname

    if not done:
        haveAtantrum(" **** I can't understand this method: ")

ClassFunction.destructor():
    for depth in range(fd_.size()):
        fd = fd_[depth] # NB: get a reference NOT a copy!
        fd.rtype.Deallocate(fd.robject.Address())
        for j in [0...fd_.values.size()-1]:
            del fd.values[j]

ClassFunction.invoke(address):
    raddr    = 0
    value    = 0
    value_   = 0

    # Keep track of objects returned by value because we need to
```

```
    # call their destructors explicitly
    condemned = vector("FunctionDescriptor")

    # Loop over each part of method
    for depth in range(fd_.size()):

        fd = fd_[depth] # NB: get a reference NOT a copy!
        execute(fd, address, raddr, value)

        if fd.simple:
            #---------------------------------------
            # Fundamental return type
            #---------------------------------------
            # This is a fundamental type returned from
            # either a regular method or:
            # 1. a bool from the isAvailable() method of a smart pointer
            # 2. a bool from the isNull() method of a smart pointer
            # This could be an isAvailable method. If so, check its
            # return value
            if fd.isAvailable:
                available = value
                if available:
                    pass
                else:
                    # The collection is not available, so return a
                    # null pointer
                    Warning("CollectionNotFound")
                    value = 0
                    break # break out of method loop

            # This could be an isNull method. If so, check its return value
            elif fd.isNull:
                null = value
                if null:
                    # The collection is not available, so return a
                    # null pointer
                    Warning("NullSmartPointer")
                    value = 0
                    break # break out of nethod loop
                else:
                    pass
        else:
            #---------------------------------------
            # Non-fundamental return type
            #---------------------------------------
            # Keep track of objects returned by value. We need to call
            # their destructors explicitly.
            if fd.byvalue: condemned.append(fd)

            if fd.pointer:
                if raddr == 0:
                    Warning("NullPointer")
                    value = 0
                    break # break out of method loop

            # Return address becomes object address in next call of
            # compound method
            address = raddr

    # Ok, we've got to the end of the chain of calls
    # Cache return value
    raddr_     = raddr
```

```
    value_      = value

    # Now, explicitly destroy objects that were returned by value
    for i in range(condemned.size()):
        # get a reference NOT a copy
        fd = condemned[i]

        # call object's destructor, but keep the memory that was
        # reserved when ClassFunction was initialized.
        fd.rtype.Destruct(fd.robject.Address(), false)

    # Finally, we return the value!
    return value_

ClassFunction.execute(fd, address, raddr, value):
    # fd       function descriptor
    # address  address of object whose method/data member is to be called
    # raddr    address of return object or value
    # value    return value

    if fd.datamember:
        raddr = datamemberValue(fd.classname, address, fd.expression)
    else:
        raddr = invokeMethod(fd, address)

    # If address is zero, bail out
    if raddr == 0: return

    # If the function does not return a fundamental type then just return
    if fd.rcode == kNOTFUNDAMENTAL: return

    # Ok the function's return type is fundamental, so map it to a double
    value = toDouble(fd.rcode, raddr)
```

C.4 Invoking the (optional) macro

A macro created with `mkmacro.py` is modeled as a class with an `analyze` method that can access all the ntuple variables before their values are committed to the tree `Events`. The code fragments below show how the macros are initialized and called by `TNM`. The macro processing is done using `CINT`. This is not as slow as one might fear because what is actually called by `CINT` is a *compiled* macro.

code fragment 1

This code fragment is called in `TNM`'s constructor. The object varmap is a map between the ntuple variable name and a `struct` of type `countvalue` that has two pointers, one that points to the values associated with the variable and another that points to the number of values. This is the object that gives the macro access to the ntuple variables. The object `indexmap` is a map from object names to the indices of the objects to be kept, should the user choose to skim objects.

```
// Create pointer "tree" and set its value to the address of the Events
   tree
string treecmd("TTree* tree = (TTree*)0x%lx");
gROOT->ProcessLine(Form(treecmd.c_str(),(unsigned long)(output.tree())));

// Create object "varmap" and set its value to the address of varmap
string mapcmd("map<string,countvalue>* "
              "varmap = (map<string,countvalue>*)0x%lx");
gROOT->ProcessLine(Form(mapcmd.c_str(), (unsigned long)(&varmap)));

// Create object "indexmap" and set its value to the address of indexmap
```

```
string imapcmd("map<string,vector<int> >* "
         "indexmap = (map<string,vector<int> >*)0x%lx");
gROOT->ProcessLine(Form(imapcmd.c_str(), (unsigned long)(&indexmap)));

// Create macro object "obj"
string macrocmd = macroname_ + string(" obj(tree,varmap,indexmap);");
gROOT->ProcessLine(macrocmd.c_str());
```

After this code has been executed, the macro object obj exists. The second code fragment shows how the methods of the macro are called. Its `beginJob` method is called before `cmsRun` enters the event loop.

code fragment 2

This code fragment is called after `TNM` has called all methods and cached their return values, but before the values have been committed to `Events`.

```
// Initialize event variables in user macro and reset indexmap
gROOT->ProcessLineFast("obj.initialize();");
// Call macro analyze method
keep = (bool)gROOT->ProcessLineFast("obj.analyze();");
```

If `keep` is true, the event is kept, otherwise it is skipped.

Bibliography

[1] V.M. Abazov et al. Observation of Single Top Quark Production. *Phys. Rev. Lett.*, 103:092001, 2009.