DRAFT
CMS Internal Note

2012/07/27
Head Id: 128481
Archive Id: 140382M
Archive Date: 2012/06/10
Archive Tag: trunk

TheNtupleMaker: A standard ntupling system for CMS analyses

Harrison B. Prosper[1] and Sezen Sekmen[2]

[1] Florida State University
[2] CERN

Abstract

We describe a standard, simple, automated ntupling system called `TheNtupleMaker` that works on all CMS EDM formats such as `RECO`, `AOD`, `PAT`, `CMG`, etc., which is based on the `ROOT` reflex mechanism. `TheNtupleMaker` allows users to configure flat ntuples via a simple configuration file and also generates an analyzer that can be used to read and analyze the resulting ntuples. Furthermore, `TheNtupleMaker` provides two mechanisms to add user-defined variables to the ntuples and skim while ntupling. The system is self-documenting and stores, in the ntuple, the most useful provenance information. Owing to the systematic and user-friendly approach, this system could save considerable of collaboration time and effort that could more fruitfully be directed to physics studies.

Contents

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 now at 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 *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 production by Dzero [1]. By any measure, the Dzero single top analyses were very sophisticated; yet almost everything was done using ntuples and the fraction of group members who made substantial contributions was atypically high.

A large number of ntuple-makers exist in CMS, and a large number existed (and exist) in CDF and Dzero. 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 yet another ntuple-maker? As the experience at Dzero 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 manpower that can be devoted to validate each of these infrastructures thoroughly.

The motivation for embarking on the `TheNtupleMaker` project in December 2009 was the absence of a *simple* and *standard* infrastructure for making ntuples having 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, `TheNtupleMaker`, which provides a *simple* and *standard* system for making ntuples, where the ntuple content can be straightforwardly designed by the user (and which additionally generates an analyzer that can be directly used for reading and analyzing the resulting ntuples). Furthermore, `TheNtupleMaker` provides mechanisms to add user-defined variables to the ntuples and skim while ntupling. `TheNtupleMaker` is designed as a *self-documenting* tool, which automatically tracks key provenance information pertaining to the EDM objects that were used in making the ntuples.

In the following, we shall first summarize the design principles of `TheNtupleMaker` in Section 2, then shortly explain the implementation of the design in Section 3. This will be followed by a detailed description of how to use `TheNtupleMaker` in Section 4, a note on provenance

[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 an embarrassing incoherence in the analysis infrastructure.

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

in Section 5, and the conclusions. Readers who would like to start using `TheNtupleMaker` immediately can proceed directly to Section 4. Technical details, which have been skipped in order not to appall the non-geek readers, can be found in the Appendices.

2 Design principles

The design principle of `TheNtupleMaker` is guided by the following advice:

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

The `ROOT` package 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 content 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, `pat::Electrons`. 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`. 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 the CMS data: it is a large collection of `float`s, `double`s, `int`s, `unsigned int`s or `bool`s each of which is associated with a simple or compound method. The purpose of `TheNtupleMaker` 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 `cmsRun` configuration file should be self-documenting.

3. The system should have an automatic way to construct the 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:

1. The ntuple uses `ROOT`'s variable-length array mechanism, rather than `STL vector`s. Instead the mapping to `vector`s is done at runtime when an ntuple is read, which is when they are needed.
2. The `cmsRun` 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`, and the method name.
4. `TheNtupleMaker` imposes a maximum compound method indirection depth of 20 [3].
5. The addition of user-defined methods is handled using a "helper" mechanism (see Section 4.6.1). In addition, variables can be added directly to the ntuple using a "macro" mechanism (see Section 4.6.2), 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 `TheNtupleMaker` 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 resist) 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.

3 Implementation

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

3.1 Introduction

`TheNtupleMaker` is an `EDAnalyzer`

```
class TheNtupleMaker : public edm::EDAnalyzer
{
public:
        explicit TheNtupleMaker(const edm::ParameterSet&);
        ~TheNtupleMaker();
```

[3] Even for CMS, this ought to be sufficient!

```
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` 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 `TheNtupleMaker`.

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

Steps (1) and (2) are performed in `TheNtupleMaker` 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.

3.2 Overview of implementation

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

```
(1)                patMuon =
                   cms.untracked.
```

```
                vstring(
(2)     "patMuon                         cleanLayer1Muons                 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 object*, or *config block* for short. A config block is a descriptor that contains the list of `pat::Muon` 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 Event Data Format (EDM) plugin. During the automatic generation of the plugin files (see Section 4), 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 simply 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 for a given `getByLabel` input tag.

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 `pat::Muons`. The second element of item (2) is the label to be used in `getByLabel` for extracting the `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 `ClassFunction`. Every `ClassFunction` object, in turn, 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 `TheNtupleMaker` 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 is available.)

In the analyze method, `TheNtupleMaker` 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 `FunctionDescriptor`s. 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 Reflex` mechanism. (See Appendix for details.)

`TheNtupleMaker` 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, `TheNtupleMaker` 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, `TheNtupleMaker` checks the validity of the pointers before attempting to use them.

Life is simple when a pointer is simple; `TheNtupleMaker` merely checks that the pointer is non-zero. If the pointer is non-zero, it is presumed that all is well and `TheNtupleMaker` 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, `TheNtupleMaker` calls the smart pointer's `isAvailable` method and checks the return value. If the return value is `true`, then `TheNtupleMaker` calls the smart pointer's `isNull` method and checks its return value. If that is `true`, then `TheNtupleMaker` calls the pointer's `get` method, which returns a simple pointer whose value is also checked. `TheNtupleMaker` 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 `FunctionDescriptor`s. The call to methods then reduces to looping over sequences of descriptors and executing the instructions they contain.

4 Using TheNtupleMaker

4.1 Location of `TheNtupleMaker`

`TheNtupleMaker` package is located in the `UserCode` area of CMS `CVS`, under the directory

```
UserCode/SusySapien/PhysicsTools/TheNtupleMaker
```

4.2 Versions

The latest version number of `TheNtupleMaker` can be found in the `TheNtupleMaker` twiki.

```
https://twiki.cern.ch/twiki/bin/view/CMS/TheNtupleMaker
```

In principle, `TheNtupleMaker` is `CMSSW` version independent.

4.3 Installation

To install version `X_Y_Z` of `TheNtupleMaker` under `CMSSW` release `A_B_C`, do the following:

```
cmsrel CMSSW_A_B_C (equivalent to: scram project CMSSW CMSSW_A_B_C)
cd CMSSW_A_B_C/src
cmsenv

mkdir PhysicsTools
```

```
cd PhysicsTools

cvs co -r vX_Y_Z -d TheNtupleMaker UserCode/SusySapien/PhysicsTools/TheNtupleMaker
cd TheNtupleMaker
scripts/initTNM.py    (or scripts/initTNM.py CMG if you have a local release
                      of the package CMGTools)

scram b clean
scram b
cmsenv
```

The build on a fast laptop takes about 10 minutes 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.
```

`TheNtupleMaker` is released with plugins for a few standard helper classes (see Section 4.6.1 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`).

You could work directly in the package directory, i.e. `PhysicsTools/TheNtupleMaker`, however, it is better and tidier to set up a user area in the `CMSSW_A_B_C/src/` directory. In the following, we assume that you wish to work in a sub-system[4] called `Ntuples` and to create a package[5] called `MyNtuple` under `Ntuples`. Assuming that you are already in the `CMSSW_A_B_C/src` directory (that is, the directory that contains your downloaded copy of `PhysicsTools/TheNtupleMaker`), you can use the script `mkpackage.py` as follows to create a skeleton of `Ntuples/MyNtuple`:

```
mkdir Ntuples
cd Ntuples
mkpackage.py MyNtuple
cd MyNtuple
scram b
```

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`.

4.4 Defining the ntuple content

The content of the ntuples are defined in a configuration file. One needs to specify the relevant *class*, *method* and *getByLabel* of each data element to be stored in the ntuple using a simple self-describing syntax. The config fragment

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

shows an example of the syntax for specifying the data elements to be included in the ntuple. It is always possible to write such a config fragment by hand, but this requires exact knowledge

[4]Subsystems are directories under `CMSSW_A_B_C/src`
[5]Packages are directories under subsystems.

of class names, class methods and getByLabel names for *i)* the events in the input files and *ii)* the `CMSSW` release that is being used. `TheNtupleMaker` overcomes this difficulty with a Graphical User Interface (GUI) that

- scans the input event file
- lists all the classes, methods and getByLabels that yield floats, doubles, ints, or bools
- allows you to select the classes, methods and getByLabels of interest
- and automatically writes a (module initialization type) config fragment, i.e. `_cfi.py` file listing the names of the selected data elements.

The GUI is usually run in the `test` directory of the package and requires a `.root` file containing the `edm` data objects from which you wish to make the ntuple. To run the GUI, go into your package `test` area, e.g.

```
cd Ntuples/MyNtuple/test
```

copy a sample of your `.root` file there, and run

```
mkntuplecfi.py &
```

The first time this Python script is run, it will scan through and analyze all the class hierarchies in the defined areas of the release, and then bring up a GUI window. In subsequent runs, the scanning will not take place, and the GUI will pop up after a few seconds. Once the GUI is up and running, do the following:

1. Select `File` $\rightarrow$ `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 the classes in it that match the defined classes of the `CMSSW` release, and that provide methods returning simple types, will be listed in the `Classes` column.
4. Select a class of interest (e.g. `vector<pat::Jet>`). 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 `getByLabel` column. The GUI lists simple methods such as `pt()` as well as compound methods such as `gsfTrack()->numberOfValidHits()`. It also lists methods, if they exist, that take arguments with simple types, such as `int`, `float`, `double`, and `std::string`. Moreover, even though the GUI does not list methods such as

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

   `TheNtupleMaker` can handle such methods and makes a valiant attempt to check that a pointer is valid for any method that returns one.
5. Select a getByLabel (e.g., `selectedPatJets`. The selected getByLabel will be highlighted.

6. Select methods of interest (e.g., `double pt(),double eta(),double phi()`). The selected methods will be highlighted. You can use the "`Find method`" area to find the methods easily. Type the *exact* name of the method in the "`Find method`" area and press `Enter`. 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 `Enter` multiple times will loop through the available methods.

7. Repeat 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 on the top to check the selection. Only classes for which at least one method and at least one getByLabel have been selected will appear here. Click on the classes to see the selected content. One can always cancel a selection by going back to the `Methods` tab and re-clicking on the names of methods and getBylabels. 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.

9. When the selection is complete, select `File` → `Save` from the menu or press the "`Save configuration file fragment`" button (the second button in the toolbar) to save the configuration fragment. The default name for the config fragment is `ntuple_cfi.py`. You should either save the config fragment directly into your python directory, or copy it to that directory after you exit the GUI.

10. Exit the GUI.

The configuration fragment

```
Ntuples/MyNtuple/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 an `ntuple_cfi.py` is given in Appendix B.

A special case that requires intervention by hand is the situation where the methods require an argument, e.g., `float bDiscriminator(std::string)`. When such a method is selected, the configuration fragment will contain the entry:

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

One needs to replace the type of the argument, 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") simpleSecondaryVertexBJetTags'
```

(In general, the alias is optional because a default name for the ntuple leaf is constructed automatically from the method name. In this example, however, we use an alias because the default name may not be suitable. Note the alias does not need to be the same as the argument; we used the same word only for illustrating the syntax.)

For methods, such as `gsfTrack()` that return pointers (smart or otherwise), it is necessary to check that the returned pointer is not null. This is handled automatically by `TheNtupleMaker`. If a method returns a null pointer, the subsequent call of the compound method will not be made.

Note: The GUI is merely an aid to get you started. It lists a large number of the methods that can be accessed by `TheNtupleMaker`. However, there are many methods accessible by `TheNtupleMaker` 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 `TheNtupleMaker` can access that method. So be experimental!

4.4.1 Special treatment of the triggers

`TheNtupleMaker` allows to access the values (0 if the trigger did not fire, 1 if it did or negative if the trigger does not exist) and prescales for triggers using `int value(trigger-name)` and `int prescale(trigger-name)`. 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, `TheNtupleMaker` supports a range and wildcard syntax that can be used to shorten what could otherwise be a long, and rather 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 "...", while a wildcard uses the standard wildcard character "*". In fact, if you feel particularly adventurous, you can use a regular expression so long as it is not overly complicated.

4.5 Making the ntuple

The next step after specifying the ntuple content is to run TheNtupleMaker over edm .root event files. Given the configuration fragment Ntuples/MyNtuple/python/ntuple_cfi.py, the next steps are:

1. Go to the package directory, in our case, to Ntuples/MyNtuple, which contains the file <packagename>_cfg.py, or in our case, MyNtuple_cfg.py.
2. Open MyNtuple_cfg.py with your favorite editor and modify the PoolSource, specifying the full name of your input event file (or files).
3. cmsRun MyNtuple_cfg.py

There will be two outcomes of this run:

- A file called ntuple.root containing the ntuple. The name ntuple.root is defined in ntuple_cfi.py

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

  and can be changed according to will.
- A directory called <packagename>analyzer, or in our case, MyNtupleanalyzer that includes an automatically created program that can be used for analyzing the ntuple (see "Analysis of the resulting ntuple"). The name MyNtupleanalyzer is also defined in ntuple_cfi.py

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

  and can be changed according to will.

  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 fragment ntuple_cfi.py (in your python directory) and create your analyzer by hand using

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

  after TheNtupleMaker has been run. The script mkanalyzer.py reads the file variables.txt (written by TheNtupleMaker) and constructs the analyzer using the information contained in that file. The file variables.txt can be created independently of TheNtupleMaker using the command

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

  For example,

  ```
  mkvariables.py myhedgehog.root Boris
  ```

will create the file `variables.txt` from which an analyzer can be created using `mkanalyzer.py`. The combination of `mkvariables.py` and `mkanalyzer.py` should work on any flat simple ntuple, not only those created by `TheNtupleMaker`.

4.6 Adding user defined variables to the ntuple

The purpose of `TheNtupleMaker` is to automate the creation of ntuples from the more than 50,000 methods and data members in `DataFormats` that return simple types. Sadly, this is sometimes not enough! Therefore, `TheNtupleMaker` also provides a *helper mechanism* to handle composite variables (e.g., `deltaphi(jet pT, MET)`). In addition, there is also a simple mechanism for calling a macro from `TheNtupleMaker` that can be used to add user-defined variables to the ntuple. We describe both of these mechanisms below.

4.6.1 Using the helper mechanism

A helper is a class that extends the interface of another class. A helper for `pat::Muon` has, automatically, 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 `TheNtupleMaker` 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, and could have been, so a helper for the `TriggerResults` object is needed (unfortunately). 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.

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 composite (i.e. there are zero or more objects in the event, like `pat::Jet`. 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, you execute

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

or for `edm::Event`, with a postfix `HelperExtra`, you execute

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

Running the script creates the helper class in the `src` and `interface` directories (the examples above will create `patJetHelper` and `edmEventHelperExtra`). After you make 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. Note however that there already exists a default `edmEventHelper` in `TheNtupleMaker` package which is designed to retrieve `edm::Event` methods such as `run()`, `event()`, `luminosityBlock()`, etc., therefore the name of the new `edm::Event` helper should be different, e.g., `edmEventHelperExtra`.

The helper mechanism can also be used for skimming objects. This will be described in Section 4.7.

4.6.2 Using the macro mechanism

`TheNtupleMaker` 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 `TheNtupleMaker` 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 you generally do not edit, lists all the ntuple variables available to you. 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 `TheNtupleMaker`. You tell `TheNtupleMaker` about your macro using the `macroName` variable,

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

The new variables will be added to the `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 4.7.

4.7 Skimming using `TheNtupleMaker` while ntupling

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

4.7.1 Skimming events while ntupling

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.

4.7.2 Skimming objects while ntupling

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

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.

4.8 Controlling messages from the `MessageLogger`

`TheNtupleMaker` uses the CMS `MessageLogger`. Therefore, it is possible to modify the frequency of messages by placing `MessageLogger` commands in the `MyNtuple_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 `MyNtuple_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 `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.

4.9 Analysis of the resulting ntuple

`TheNtupleMaker` provides an automatically generated analyzer for analyzing the ntuples. The analyzer:

- Reads the contents of the ntuple.
- Assigns all the contents (the branches) to variables, either simple or vector variables as appropriate, with automatically generated names.

If you don't like the automatically generated names, you can edit the file `variables.txt` in your test directory. This file is created every time `TheNtupleMaker` is run. You can edit the file and replace all the variable names that annoy you with ones that don't. 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 ntuple leaf, the third is the variable name, and the last field is the maximum item count for the variable. Feel free to change the variable name fields to names you like. Then (re)create your analyzer with:

```
mkanalyzer.py analyzer
```

(An asterisk at the end of an entry in `variables.txt` indicates that this variable is a leaf counter.) Automatically saves all histograms generated by your analyzer to a `.root` file.

There are two versions of the analyzer, both of which work independently of the `CMSSW` framework, requiring only `ROOT` as a prerequisite:

- a C++ version.
- a Python version.

Both versions can, however, work within the `CMSSW` framework if needed. For the C++ version, simply copy the source file and its header to your `bin` directory and modify the `BuildFile` in `bin` appropriately.) For the Python version, just un-comment the import statement

```
from PhysicsTools.TheNtupleMaker.AutoLoader import *
```

in `analyzerlib.py`. 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 modify this file. (Indeed, it is better not to because you may wish to swap it for an updated header.) 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` gets the variables from the ntuple and loops over the events. This is the code to be modified. You can book histograms and write analysis code at the places pointed out by inline comments. Feel free to add branches, trees, etc. to the output root file.
4. Compile using `make` (this should create the executable `analyzer`).
5. The executable `analyzer` is run as follows:

   ```
   ./analyzer <name of file listing the ntuple file names, filelist.txt by default>
   ```

   The `filelist.txt` is a simple text file that lists the names of the ntuple root files to be read, 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 all the histograms you created, including any directory structure you may have imposed on them.

5 `TheNtupleMaker` and provenance

It is rightly considered particularly important in CMS to have access to the so-called *provenance* information of a given sample, which allows us to know about the history of the sample, and how the contents of the sample are constructed. `TheNtupleMaker` 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 necessary:

- The variable names in the `Event` tree are constructed, by defalt, as

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

  and hence reflect the exact EDM origins of the variables.

- 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` which was used for creating the ntuple, and as explained above, this config contains the EDM information on all variables stored in the ntuple, i.e., their classes, getByLabels and methods.

6 Conclusions

We presented `TheNtupleMaker`, a standard ntupling system for CMS analyses. `TheNtupleMaker` is a system based on the `ROOT` reflex mechanism, which

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

In a time that is so exciting for LHC physics, it is crucial to have tools that help to produce physics results as efficiently as possible, and help save our time and effort for thinking and performing physics itself. `TheNtupleMaker` was designed with this aim, in order to serve CMS as a systematic and user-friendly ntuple-making system and its analyzer, which can easily be adapted by any analysis – and it is already adapted by various CMS analyses. `TheNtupleMaker` is intended to be a tool that can be used easily by any analyst, without requiring extensive computing skills, and hence makes it easier for more physicists to contribute effectively to improving our physics output.

Acknowledgements

We would like to thank the many people, who, like us, believe that physics should be about physics, that the tools we use for doing physics should not be overly complicated beyond absolute necessity, and who therefore helped us test and improve `TheNtupleMaker`. Many thanks to Jeff Haas for his support since the very beginning; to Joe Bochenek and Lukas Vanelderen for constantly using `TheNtupleMaker` and giving valuable feedback; to our SUSY RA2b colleagues Don Teo, Josh Thompson, Ben Kreis, Harold Nguyen and Sudan Paramesvaran; to Nadja Strobbe and Piet Verwilligen; to Supriya Jain; and to our EXO dijet colleagues Maurizio Pierini, Francesco Santanastasio and Maxime Gouzevitch for adopting `TheNtupleMaker` in their analyses and making many useful suggestions. We also thank Maria Spiropulu for supporting and spreading the idea.

A Initializing TheNtupleMaker with initTNM.py

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 one does not exist.
3. Create a soft link

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

   that points to

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

   This makes the `Python` modules in `TheNtupleMaker`'s `python` directory available to the scripts to be executed by `initTNM.py`.
4. Recursively scan the directories (using `mkclassmap.py`)

   ```
   AnalysisDataFormats/*
   DataFormats/*
   SimDataFormats/*
   FWCore/Framework
   FWCore/FWLite
   FWCore/MessageLogger
   FWCore/ParameterSet
   FWCore/Utilities
   FWCore/Common
   PhysicsTools/TheNtupleMaker
   ```

   for C++ header files and extract from them fully scoped class names (e.g., `pat::Muon` rather than `Muon`, which is ambiguous) of classes that seem potentially of interest. Then create the file `python/classmap.py` containing a `Python` map between fully scoped class names and headers.
5. 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`.
6. 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 TheNtupleMaker configuration file

The configuration file fragment `ntuple_cfi.py`, which resides in the `python` directory of a package, contains the instructions to control `TheNtupleMaker`. In particular, `ntuple_cfi.py` specifies what methods are to be called by `TheNtupleMaker` to create an ntuple. As described in Section 4, 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 `TheNtupleMaker`, is to be dynamically loaded into and run by `cmsRun`. **ntupleName** (2), **analyzerName** (3), and **buffers** (4a) - (4b) are the main attributes of `TheNtupleMaker`.

(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 `TheNtupleMaker` 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, `TheNtupleMaker` 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

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

Note: **param** is a reserved keyword that tells `TheNtupleMaker` 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 `TheNtupleMaker` 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 `TheNtupleMaker`.

C.1 The `Reflex` mechanism

`Reflex` is a set of C++ classes, distributed with `ROOT` (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 1) 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, 2) `ROOT::Reflex::Member` that represents a class data member or method, and 3) `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 `TheNtupleMaker`, the `Reflex` mechanism is encapsulated in the (`TheNtupleMaker`) 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 `TheNtupleMaker`! 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 = ""
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
            fd.rname = fd.classname # the same class name as current smart pointer
            expr2 = "isNull()" + delim + expr2

        elif fd.isNull:
            # This is an isNull method, so insert a call to get
            fd.rname = fd.classname # same classname as current smart pointer
            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 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 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 loop

            # Return address becomes object address in next call of compund 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 being 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 `TheNtupleMaker`. 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 `TheNtupleMaker`'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 `TheNtupleMaker` 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.

References

[1] V. M. Abazov *et al.* [D0 Collaboration], "Observation of Single Top Quark Production," Phys. Rev. Lett. **103** (2009) 092001 [arXiv:0903.0850 [hep-ex]].