



BNL-98504-2012-CP

***Toolkit for data reduction to tuples for the ATLAS
experiment***

Scott Snyder¹ and Attila Krasznahorkay²

(For the ATLAS Collaboration)

¹ Brookhaven National Laboratory, Upton, NY, 11973, USA

² Physics Department, New York University, 4 Washington Place,
New York, NY, 10003, USA

Presented at the Computing in High Energy Physics (CHEP12) Conference
New York University, New York, NY, USA
May 21 – 25, 2012

September 2012

Physics/High Energy/ATLAS

Brookhaven National Laboratory

U.S. Department of Energy

Notice: This manuscript has been authored by employees of Brookhaven Science Associates, LLC under Contract No. DE-AC02-98CH10886 with the U.S. Department of Energy. The publisher by accepting the manuscript for publication acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

This preprint is intended for publication in a journal or proceedings. Since changes may be made before publication, it may not be cited or reproduced without the author's permission.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or any third party's use or the results of such use of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof or its contractors or subcontractors. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Toolkit for data reduction to tuples for the ATLAS experiment

Scott Snyder¹ and Attila Krasznahorkay²
(For the ATLAS Collaboration)

¹ Brookhaven National Laboratory, Upton NY, 11973, USA

² Physics Department, New York University, 4 Washington Place, New York, NY, 10003, USA

E-mail: snyder@bnl.gov

Abstract. The final step in a HEP data-processing chain is usually to reduce the data to a ‘tuple’ form which can be efficiently read by interactive analysis tools such as ROOT. Often, this is implemented independently by each group analyzing the data, leading to duplicated effort and needless divergence in the format of the reduced data. ATLAS has implemented a common toolkit for performing this processing step. By using tools from this package, physics analysis groups can produce tuples customized for a particular analysis but which are still consistent in format and vocabulary with those produced by other physics groups. The package is designed so that almost all the code is independent of the specific form used to store the tuple. The code that does depend on this is grouped into a set of small backend packages. While the ROOT backend is the most used, backends also exist for HDF5 and for specialized databases. By now, the majority of ATLAS analyses rely on this package, and it is an important contributor to the ability of ATLAS to rapidly analyze physics data.

1. Introduction

The final stage for most HEP analyses is often works from a ‘tuple’ data format which can be easily manipulated with interactive analysis tools such as ROOT [1]. A tuple can generally be thought of as a two-dimensional table where the rows are events and the columns are variables describing the event. These variables can be of either simple or more complicated data types; in particular, events will typically contain various collections of objects (electrons, muons, etc.), so the tuple variables for them will also be collections.

These tuple data sets are typically the responsibility of the group working on the analysis. However, in a large experiment such as ATLAS, this rapidly leads to having many different group-specific codes producing group-specific tuple formats. Not only does this result in substantial duplicated effort in implementation and validation, but the fact that each tuple has independent conventions and naming makes communication between groups difficult, and hinders the provision of common tools that work from the tuple format.

On the other hand, however, different physics analyses do have differing requirements. Attempting to produce a single tuple format that has sufficient information for all analyses typically results in a tuple that’s too big to satisfy anyone.

Therefore, ATLAS has pursued a toolkit approach: ATLAS provides a set of common tools which groups doing physics analyses can combine together, and possibly customize, to form tuples appropriate for their analysis. The toolkit also can be easily extended, to allow adding

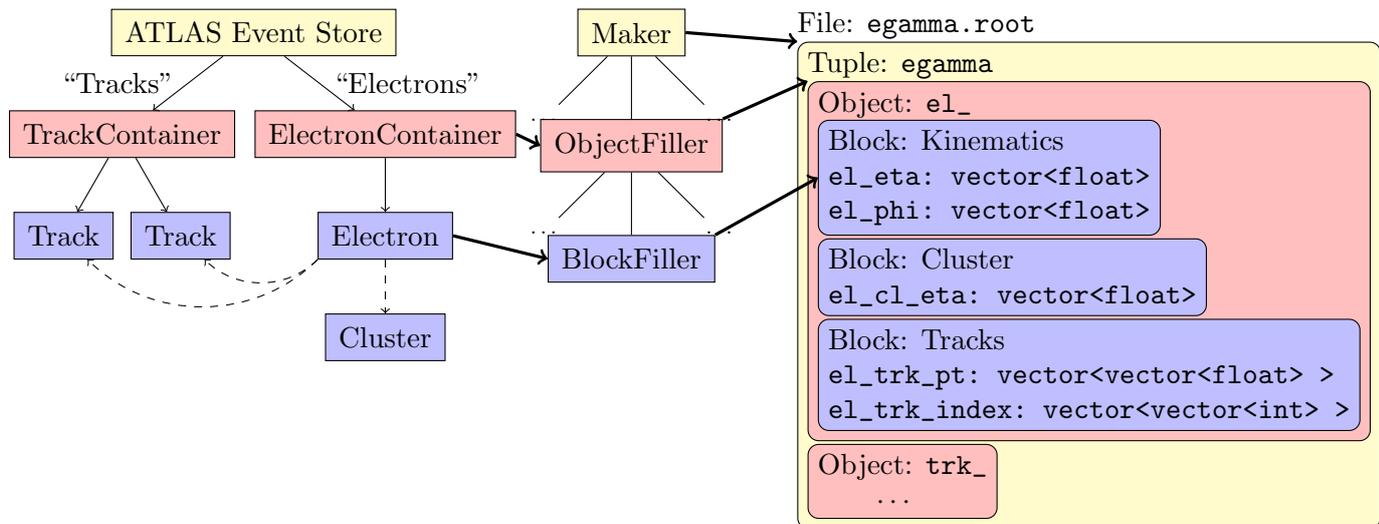


Figure 1. Overview of tuple making. The tools in the middle process the contents of the ATLAS event store, on the left, to the tuple structure on the right.

information specific to a particular analysis. This approach preserves flexibility while sharing most of the development and validation effort; further, tuples made using the toolkit will share a common set of conventions and names, facilitating sharing between groups.

This note is organized as follows. Section 2 gives an overview of the components of the toolkit. Section 3 discusses how these components are combined to produce a complete tuple. Section 4 outlines the tools that need to be written in order to add new kinds of data to the tuple. Section 5 discusses the use of type-generic tools in the toolkit. Section 6 discusses the interface used to allow writing tuples in multiple formats. Section 7 outlines utilities that ATLAS makes available to aid in reading tuples. Finally, section 8 includes some comments on the use of the tuples in ATLAS, and section 9 is the summary.

2. Overview

The goals for the design of the toolkit included the following:

- The tuple should be usable for analysis with minimal runtime support. This generally implies a ‘flat’ tuple structure, using only standard C++ types. However, the toolkit should not preclude more structured types as well.
- Variables in the tuple should be grouped into related blocks at a relatively fine level of granularity. Each block is associated with a ‘level of detail’; blocks can be selected by choosing a level of detail or individually.
- The tuple should be easily extensible by user code without requiring changes to core packages.
- It should be easy to change the underlying method used to store the tuple data.

The toolkit runs in the context of the ATLAS offline software framework [2–4]. This framework is based on a collection of loosely-coupled, dynamically-loaded components which communicate via a “blackboard”-style event store, from which objects can be retrieved based on type and a name. The tuple toolkit provides components that access information from the event data store and convert it to a tuple format.

An sketch of this is shown in figure 1. The toolkit produces files containing one or more tuples; each tuple is managed by a “Maker” component, which the offline framework will call

for every event. The tuple is divided into a set of “objects,” which correspond to objects from the ATLAS event store. Objects are managed by “ObjectFiller” tools, and the variables within them share a common prefix. The variables within an object are grouped into “blocks,” which define the granularity at which variables can be selected to be included in or omitted from the tuple. Each block is managed by a “BlockFiller” tool, and is associated with a small integer “level of detail.” During configuration, all blocks with a certain level of detail or less may be selected; blocks may also be individually selected or excluded.

So, for each event, the framework calls the Maker component for each tuple being built. The Maker then calls in turn each of the ObjectFiller tools. Each ObjectFiller tool will then obtain its input, usually from the event data store. The input can either be a single object or a container of objects. In the former case, the ObjectFiller simply passes it in turn to each of the BlockFiller tools. In the latter case, the ObjectFiller tool loops over the contents of the container and calls the BlockFiller tools for each; it is also responsible for building a vector for each of the variables containing the results. Note that a BlockFiller tool only ever deals with a single object; handling of containers is done by the caller. This makes it easier to reuse BlockFiller tools in different contexts.

An important concept is that of associations. An association maps a *source* object to one (single association) or a set of (multiple association) *target* objects. The target objects may be contained within the source object, may be referenced by the source object, or may be constructed dynamically by the association. An association may be represented in the tuple either by having the target objects “contained” within the source objects, or by storing within the source objects the indices of the target objects in another part of the tuple.

Some examples may make this clearer. Consider first figure 2. An electron has associated with it a cluster of cells in the calorimeter. The cluster has some energy, but the electron’s energy may in principle be different (due to corrections or to incorporating information from tracking). So we want to store both. Assume that the electron and cluster share a common interface for retrieving kinematic information, so that we can use the same BlockFiller tool for both. We configure the toolkit starting with a Vector ObjectFiller, a tool which records containers as vectors. This will retrieve the container “Electrons” from the event data store and pass its elements to the contained BlockFiller tools; each variable created by these BlockFiller tools will be wrapped in a vector, with one entry for each element in the ObjectFiller’s input container. The ObjectFiller will also add the prefix “e1_” to all variables created by those BlockFiller tools. This ObjectFiller then holds a BlockFiller tool to fill the transverse momentum of the electron itself; the tuple variable this ends up producing will be called “e1_pt” and will have type “vector<float>”. A second BlockFiller is added to handle the association to the cluster. This tool holds a single Association tool (going from an electron to its cluster) and a list of BlockFiller tools; here this list contains another momentum filler tool. The association BlockFiller will also add the additional prefix “c1_” to the variables it creates. We thus have another variable “e1_c1_pt,” with entries also corresponding to the elements of the electron container.

It is also possible to have multiple associations; for example, an electron may in principle have multiple tracks associated with it. This is illustrated in figure 3. In comparison to the previous case, the Association tool is replaced with one that associates from an electron to a set of tracks, and the BlockFiller tool holding it is replaced with a MultiContainedAssociation tool. This makes one entry in the vectors for each target object; the variables that are not part of the association are duplicated.

More useful in practice is to express the contents of the target object as nested vectors rather than as duplicate rows; see figure 4. This is accomplished simply by changing the association BlockFiller tool to ContainedVectorMultiAssociation, which wraps the target variables in an additional level of vectors. The e1_trk_pt variable will now have type vector<vector<float> >, with a vector of values for each track for each element in the electron

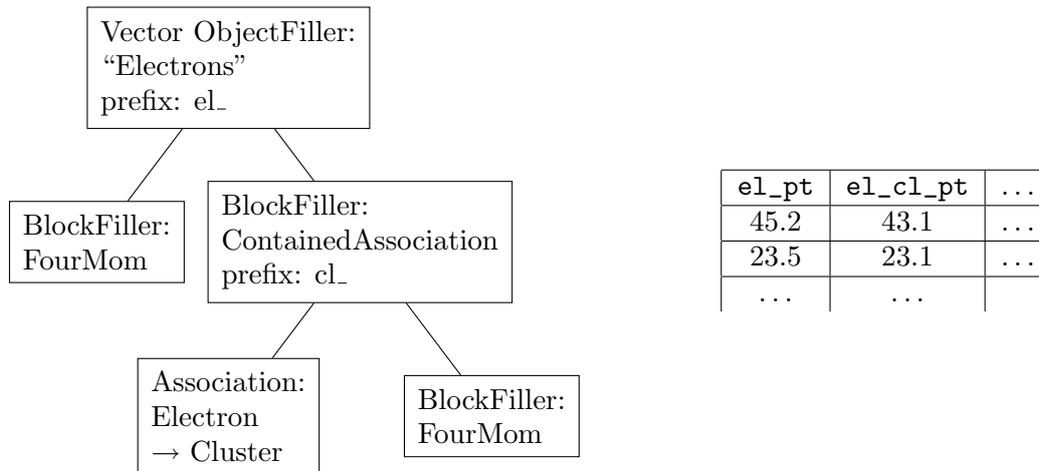


Figure 2. Example of contained single association. Left: diagram of toolkit components. Right: Example of the produced tuple.

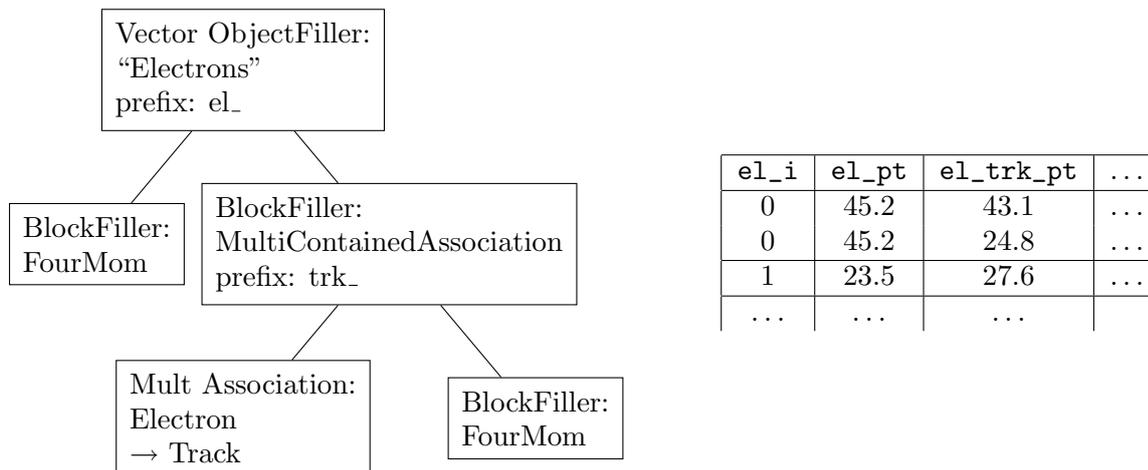


Figure 3. Example of contained multiple association. Left: diagram of toolkit components. Right: Example of the produced tuple.

container.

An association can also be represented by storing an index into another object of the tuple. For example, suppose in figure 5 that separate track objects are stored in the tuple, with a prefix of “trk_”. We can then use the special BlockFiller tool IndexFiller, which stores the index of the target track object within the trk_ object of the tuple.

Finally, in addition to the per-event data stored in the tuple, it is often useful to store additional metadata applicable to the entire data sample. This can include information about the processing history or beam conditions for the sample, as well as information about the structure of the tuple itself. For ROOT tuples, the metadata is saved as objects in a separate directory within the ROOT file alongside the tuple itself.

Here is a summary of the types of components in the toolkit.

Maker Top-level component responsible for building a tuple. Contains a list of ObjectFiller tools, over which it iterates for each event.

Getter A tool to abstracts the process of locating the input to an ObjectFiller tool. Can

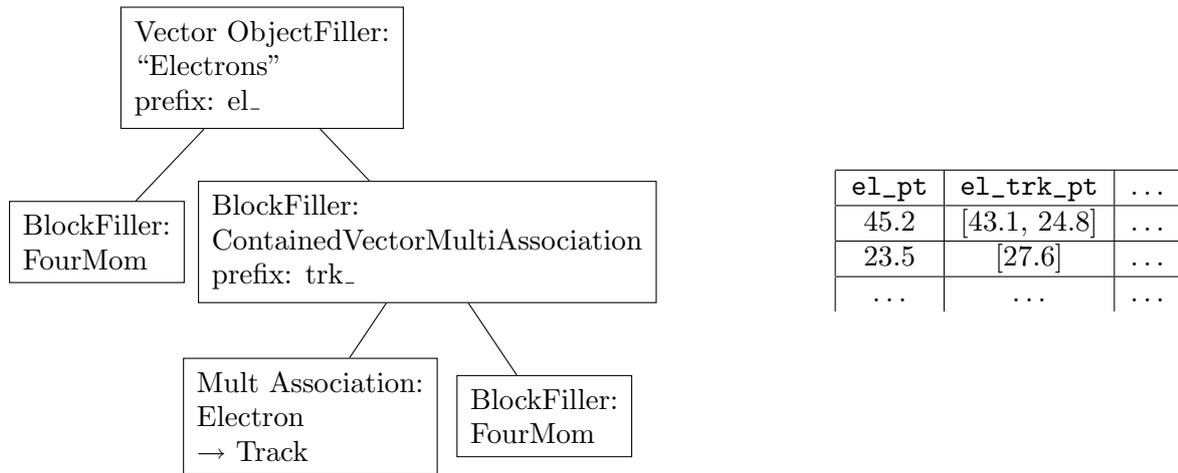


Figure 4. Example of contained multiple association, stored as nested vectors. Left: diagram of toolkit components. Right: Example of the produced tuple.

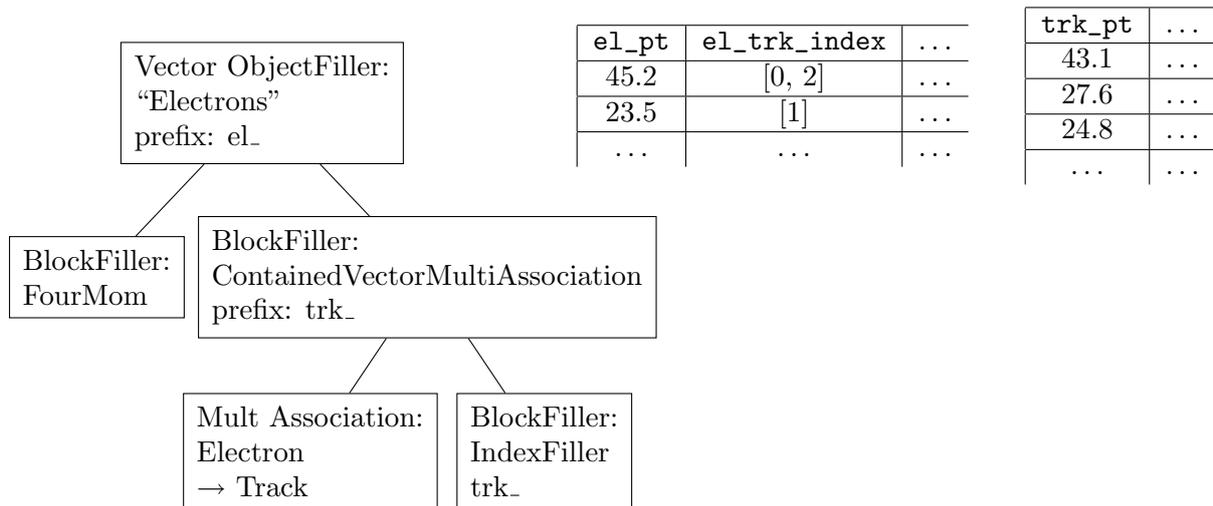


Figure 5. Example of indexed multiple association. Left: diagram of toolkit components. Right: Example of the produced tuple.

retrieve either a single object or a container of objects; in the latter case, it also provides an interface to iterate over the objects within the container. Standard Getter tools are provided for the usual case of retrieving objects from the event data store; however, there are some special cases for which alternate implementations are useful.

ObjectFiller Handles filling all variables for one tuple object. It contains a Getter tool, which is used to obtain the input object, and a list of BlockFiller tools. Several varieties of ObjectFiller tool are provided. One simply passes the input object directly to the BlockFiller tools. Another expects the input object to be a container. It creates a vector for each variable and loops over the contents of the container. It passes each element to the BlockFiller tools, telling them to fill in the appropriate elements of the vector. A third variety does not take any object as input: this is used for some blocks for which the input does not come directly from the event data store (for example, trigger decisions).

BlockFiller Copies data from an input object to a set of output variables. This is the type of

tool one needs to write in order to add support for a new object type. A BlockFiller tool that expects input of type T will derive from `BlockFillerTool<T>`; however, it is possible to write a generic tool that can accept any type as input.

Some special BlockFiller tools are used to support associations as described above. They contain lists of child BlockFiller tools that are applied to the target(s) of the association.

Association Tools deriving from `SingleAssociationTool<FROM_T,TO_T>` take an object of type FROM_T as input and return an object of type TO_T. Tools deriving from `MultiAssociationTool<FROM_T,TO_T>` are similar, but provide an interface to iterate over a set of objects of type TO_T.

MetaData As discussed above, additional information may be written alongside the tuple after the completion of event processing. This is accomplished by registering a MetaData tool with the Maker. The tool will be called when event processing is complete.

3. Configuration

This section describes the configuration of the toolkit. Like all parts of the ATLAS offline framework, the toolkit is configured using Python scripts.

Here is an example of a function to define a tuple. This is similar to what a group doing a physics analysis would maintain to define a tuple for that analysis (albeit much simpler).

```
1 def myTuple():
2     tuple = MakerAlg ('mytuple', file = 'myfile.root')
3     tuple += ElectronTupleObject (3)
4     tuple += TrackTupleObject (1)
5     tuple += ElectronTupleObject (0,
6                                     sgkey = 'myEles',
7                                     prefix = 'myel_',
8                                     include = ['Shape'])
9     return tuple
```

The tuple is created by the call in line 2. The first argument gives the name of the tuple being created; the name of the file in which it is contained may also be specified. By default, a ROOT tuple is made; however, a different sort of tuple may be made instead by passing to the Maker an alternate tool to use to create the tuple.

An object is added to the tuple in line 3. The argument is the requested level of detail, with larger numbers representing more detailed information. The example here requests that all blocks with a level of detail of 3 or less be included. The toolkit also needs to know the name of the object to retrieve from the event data store and the prefix to add to this object's variables. Here we use the defaults that were specified when the object was defined (see below). Line 4 shows adding another object with a different level of detail. Line 5 shows adding another electron object. This time, rather than taking the defaults, the code requests the electron named "myEles" from the event data store, and the prefix to be used in the tuple is set to "myel_". This object is requested with a level of detail of 0; however, the block named "Shape" is also explicitly requested.

The next example shows how to define an object to add to a tuple. These parts of the configuration are generally maintained by groups responsible for specific physics objects: electrons, muons, etc.

```
1 # Create a tuple object --- think of it like a class.
2 # Arguments are the name of the C++ container type, the default name
3 # of the object in the event store, and the default prefix within
```

```

4 # the tuple.
5 ElectronTupleObject = make_SGDataVector_TupleObject ('ElectronContainer',
6                                                     'Electrons',
7                                                     'el_')
8
9 # Define some blocks. Arguments are the level of detail, the block name,
10 # and the BlockFiller tool.
11 ElectronTupleObject.defineBlock (0, 'Kinematics', FourMomFillerTool,
12                                 WriteMass = False)
13 ElectronTupleObject.defineBlock (1, 'Shape', EleShapeFillerTool)
14
15 # Add cluster kinematics by associating to a cluster
16 # and adding another kinematics block.
17 ElClusterAssoc = SimpleAssociation (ElectronClusterAssocTool, prefix = 'cl_')
18 ElClusterAssoc.defineBlock (1, 'ClusterKin', FourMomFillerTool)
19
20 # Associate to set of tracks. Add both track momenta
21 # directly and indices into track list in the tuple.
22 TrkClusterAssoc = ContainedVectorMultiAssociation (ElectronTrackAssocTool,
23                                                    prefix='trk_')
24 TrkClusterAssoc.defineBlock (1, 'TrackKin', FourMomFillerTool)
25 TrkClusterAssoc.defineBlock (1, 'TrackIndex', IndexFillerTool, Target='trk_')

```

The object is defined in line 5. Here, “`ElectronTupleObject`” can be thought of as being analogous to a class, which is instantiated when it gets added to a tuple. The function `make_SGDataVector_TupleObject` constructs a tuple object that retrieves a container from the event store and fills vector variables in the tuple with the elements. The arguments are the name of the C++ type of the object to be retrieved from the event store, the default name of the object to be retrieved, and the default prefix to use for this object in the tuple.

Two blocks are defined starting at line 11, using the `defineBlock` method. The arguments are the level of detail for the block, the name of the block, and the `BlockFiller` tool used to fill the block. Any additional arguments are passed through to the `BlockFiller` tool; for example, the argument “`WriteMass = False`” is passed to the `Kinematics BlockFiller`. (Such settings may be overridden when the tuple object is instantiated by passing in arguments like “`ElectronTupleObject (3, Kinematics_WriteMass = False)`”.)

A single contained association is set up by `SimpleAssociation`, as shown in line 17. Here the arguments are the `Association` tool and an optional prefix to add to variables created by this association. `SimpleAssociation` returns another tuple object; `defineBlock` can then be used to create blocks taking as input the target object of the association.

Similarly, line 22 sets up a multiple association. `ContainedVectorMultiAssociation` puts the results of the association in nested vectors, as in figure 4. As before, one uses `defineBlock` to create variables using the target objects of the association. The special `BlockFiller` `IndexFillerTool` is used to record the index of the object within another object in the tuple; the “`Target`” argument refers to the prefix of that other object. It is also possible to give `IndexFillerTool` a list of targets; in that case, the index of the target in which the object is found may also be saved.

In the usual case, the level of detail argument passed to `defineBlock` is a small integer. But a function may be specified in order to implement more complicated logic. For example, the fragment below demonstrates defining a block that will be present only if an argument “`TrackIndex_Target`” is specified when the object is added to the tuple.

```

1 # Level-of-detail function for TrackIndex.
2 # The arguments are the requested level of detail and the dictionary
3 # of arguments to be passed to the BlockFiller (which the function
4 # may modify, if desired).
5 def trkAssocLevel (reqlev, args):
6     return reqlev >= 1 and args.get ('Target')
7
8 # Define a block with a level-of-detail function.
9 TrkClusterAssoc.defineBlock (trkAssocLevel,
10                             'TrackIndex', IndexFillerTool, Target='')

```

4. Writing BlockFiller and Association tools

The most common tools that one need to write when using the tuple toolkit are BlockFiller tools and Association tools. Examples of these tools are given below. (Some details such as namespace assignments have been omitted for these examples.)

4.1. BlockFiller tools

A BlockFiller tool copies data from a C++ object into the tuple. Here is a simple example of filling variables from a hypothetical four-momentum class. Further notes are given after the example.

```

1 // Example block filler tool, taking as input a hypothetical 'FourMom' class.
2 struct FourMomFillerTool
3     : public BlockFillerTool<FourMom>
4 {
5     private:
6         // Variables being filled. The pointed-to type sets the type
7         // of the tuple variable.
8         // Class types may also be used.
9         float *m_pt, *m_eta, *m_phi, *m_m;
10
11        // Tool properties.
12        bool m_WriteMass;
13
14
15    public:
16        // Constructor. Called by the component framework, not by user code.
17        // The arguments are used by the framework, and should just be
18        // passed through to the base class.
19        FourMomFillerTool (const std::string& type,
20                          const std::string& name,
21                          const IInterface* parent)
22            : BlockFillerTool<FourMom> (type, name, parent)
23        {
24            // Declare a tool property.
25            declareProperty ("WriteMass", m_WriteMass = false,
26                            "Should the mass be written?");
27        }
28

```

```

29 // Called once at the start of the first event to declare variables to fill.
30 virtual StatusCode book()
31 {
32     CHECK( addVariable ("pt", m_pt, "Transverse momentum" ) );
33     CHECK( addVariable ("eta", m_eta, "Pseudorapidity" ) );
34     CHECK( addVariable ("phi", m_phi, "Polar angle" ) );
35     if (m_WriteMass)
36         CHECK( addVariable ("m", m_m, "Mass" ) );
37     return StatusCode::SUCCESS;
38 }
39
40 // Called for each object. The tuple toolkit is responsible
41 // for setting the pointers appropriately before each call.
42 // On entry, the contents of each variable will have been initialized to 0.
43 virtual StatusCode fill (const FourMom& p)
44 {
45     *m_pt = p.pt();
46     *m_eta = p.eta();
47     *m_phi = p.phi();
48     if (m_WriteMass)
49         *m_m = p.m();
50     return StatusCode::SUCCESS;
51 }
52 };

```

Line 2 defines the class. A BlockFiller tool taking a specific type T as input should derive from BlockFillerTool<T>. It is also possible to declare a tool which can accept one of a set of types, by using the form BlockFillerTool<Types<T1, T2> > (in this case, the tool should declare multiple overloads for the fill() method described below). As a special case, if T is void, this means that the tool does not expect any input (and in this case, the fill() method should be declared with no arguments).

Line 9 defines member variables corresponding to each tuple variable to be filled; the type of the member variables should be a pointer to the desired type. (This is what sets the type of the variables in the tuple.) Tools can also have additional properties that are managed by the component framework and are set during initialization. Line 12 declares a member variable corresponding to one such property.

Line 23 is the definition of the constructor. In the ATLAS component framework, the tool constructors are not called directly by user code, but rather by the framework. From the point of view of the tool, the constructor arguments are simply boilerplate to pass through to the base class. The constructor is, however, where tool properties are defined to the component framework; here, we define a flag that will control whether or not we should write the invariant mass to to the tuple.

Line 31 shows the book method. This is called once by the tuple toolkit and should make one addVariable call for each tuple variable to be filled. The addVariable method has two required arguments: the name of the variable (to which the toolkit may then add a prefix) and a member variable that is a pointer to the type of the variable. An optional third arguments gives a documentation string for the variable; this is stored in the written tuple. An optional default value for the variable may also be given to addVariable. (The CHECK macro used here is an ATLAS idiom to check the return status of a call.)

Line 44 shows the `fill` method. This is called once for every object, and should copy data from that object into the locations pointed to by the member variables that were passed to `addVariable`. The toolkit is responsible for setting these pointers appropriately before the call to `fill`; the objects pointed to are guaranteed to be cleared to zero (or the default value given to `addVariable`, if it was specified) at the entry to `fill`.

4.2. Single Association tools

A single Association tool maps from one object to another.

```
1 // Example single association tool.
2 class ElectronClusterAssociationTool
3   : public SingleAssociationTool<Electron, Cluster>
4 {
5 public:
6   // Constructor omitted...
7
8   // Perform the association. May return null.
9   virtual const Cluster* get (const Electron& p)
10  {
11    return p.cluster();
12  }
13 };
```

A single Association tool derives from `SingleAssociationTool<FROM_T, TO_T>` (line 3). The `get` method (line 10) should take an object of type `FROM_T` and return a pointer to `TO_T`. A null pointer may also be returned (in which case any BlockFillers depending on this association will not be called, and their variables left with their default values). Usually the `get` method is very simple, as in this example. However, it is also possible for the `get` method to construct the resulting object dynamically. In this case, the tool should also define a method `releaseObject (const TO_T&)`; this will be called when the toolkit is finished with the object and it can be deleted.

4.3. Multiple Association tools

A multiple Association is somewhat more complicated than a single Association because the Association tool must also be able to act as an iterator over the results of the association. Here's an example.

```
1 // Example multiple association tool.
2 class ElectronTrackAssociationTool
3   : public MultiAssociationTool<Electron, Track>
4 {
5 private:
6   Electron::TrackConstIterator m_beg, m_end;
7
8 public:
9   // Constructor omitted...
10
11  // Prepare to start an association starting from an object.
12  virtual StatusCode reset (const Electron& p)
13  {
```

```

14     m_beg = p.tracks_begin();
15     m_end = p.tracks_end();
16 }
17
18 // Return the next target in the association.
19 // Return 0 at the end of the iteration.
20 virtual const Track* next ()
21 {
22     if (m_beg == m_end)
23         return 0;
24     return *m_beg++;
25 }
26 };

```

A multiple Association tool derives from `MultiAssociationTool<FROM_T, TO_T>` (line 3). Rather than a single `get` method, the association is done by first calling the `reset` method (line 13), passing in the source object, and then calling `next` (line 21) repeatedly until it returns null to signal the end of the association. In general, this means that the tool will need to save the state of the iteration (as in line 6).

5. Generic tool implementation

Some of the tools used to make tuples necessarily depend on the type of the object being processed, for example most `BlockFiller` and `Association` tools. Others, such as `ObjectFiller` tools, pass around pointers to data objects but never look inside them. However, to be correct according to C++ type rules, these tools would need to be templated on the type being processed. Each instantiation of these template classes would then have to be made known to the component framework. This entails significant extra effort to allow the toolkit to work with a new type, and results in duplication of code at runtime.

But simply using a generic pointer and keeping track of the object type is not sufficient. For example, if one has an `Electron` object that derives from a `FourMom` class, one would like to be able to add to the electron tuple object a `BlockFiller` that takes a `FourMom` as input. Thus, the toolkit also needs to understand the inheritance relations between classes.

This information is not available using the standard C++ run-time type identification. However, the ATLAS event data model does make such information available. This relies on using a special macro to declare inheritance relations. In this example:

```

1     struct B {};
2     struct D : public B {};
3     SG_BASE (D, B);

```

the `SG_BASE` macro declares that `D` derives from `B`. (There are other forms of the macro for multiple and virtual derivation.)

One can then use the `BaseInfoBase` interface to test classes for inheritance relationships and to convert pointers:

```

1 // Given a generic pointer to FROM_TYPE, convert it to a pointer to TO_TYPE.
2 const void* convert (const void* p,
3                     const std::type_info& from_type,
4                     const std::type_info& to_type)
5 {
6     const BaseInfoBase* bib = BaseInfoBase::find (from_type);

```

```

7   if (bib && bib->is_base (to_type))
8       return bib->cast (p, to_type);
9   return 0;
10  }

```

The container type usually used in the event data store is `DataVector<T>`. This acts like a `vector<T*>`, but has an additional feature that if a macro “`DATAVECTOR_BASE (D, B);`” has been given, then `DataVector<D>` will derive from `DataVector`. Thus, the containers have an inheritance hierarchy that mirrors that of the elements. The same `BaseInfoBase` interface described above works for such `DataVector` classes as well. Further, an interface is provided to access the element pointers of the `DataVector` without having to know the specific type of the container.

We can now outline how this is used in the tuple toolkit. During job initialization, a `configure` method is called on each `ObjectFiller` tool. Recall that each of these tools has a `Getter` tool that abstracts the process of retrieving the input object from the event data store (or possibly somewhere else). The `ObjectFiller` tool calls a method of the `Getter` interface that returns the type of object that it retrieves. It then calls a `configure` method for each `BlockFiller` tool. These tools then compare this type to the type they expect to receive as input and prepare to do a type conversion if the conversion is legal, or produce an error otherwise.

During event processing, the `ObjectFiller` calls the `fillUntyped` method, which is implemented in the `BlockFillerTool<T>` template class. This takes the argument as a generic pointer, performs the conversion to a `T*`, and then calls the `fill` method of the derived class.

This approach allows type checking to be done during job initialization, before events are processed. Also, much of the work needed to set up the type conversions can be done during initialization, reducing the overhead during event processing.

6. Alternate storage formats

While by default, tuples are written in ROOT format, the toolkit was designed so that other formats can also be used. The toolkit defines a simple abstract interface for writing data to a tuple, with these essential methods:

```

1   virtual StatusCode addVariable (const std::string& name,
2                                   const std::type_info& ti,
3                                   void* & ptr,
4                                   const std::string& docstring = "",
5                                   const void* defval = 0);
6   virtual StatusCode capture();
7   virtual StatusCode clear();
8   virtual StatusCode addMetadata (const std::string& key,
9                                   const void* obj,
10                                  const std::type_info& ti);
11

```

The `addVariable` method (line 1) is used to declare a new variable to be written to the tuple. It takes the variable name, the variable type (as a C++ `type_info`), and a reference to a pointer. The tuple implementation should initialize the pointer appropriately. Optional arguments allow specifying a documentation string for the variable and a default value for it.

The `capture` method (line 6) takes the current values of all the tuple variables and makes a new persistent tuple entry from them, while the `clear` method (line 7) resets all variables

to zero (or their specified default values). Finally, the `addMetadata` method allows adding an arbitrary named object to be written along with the tuple.

In addition to ROOT, an interface is also available for writing tuples in HDF5 format [5]. ATLAS also plans to use this mechanism for creating the “tag” database: this is a relational database that holds basic information about each event, to allow rapid selection of events of interest [6].

7. Reading tuples

The tuples produced by the toolkit, with the default ROOT backend, can be read by ROOT with no additional software, and many analyses on ATLAS do start from this point. However, it is often more convenient to be able to have a more object-oriented view of the data. This is provided by the `TupleReader` package.

When a tuple is written, extra metadata is saved that records the association between variables in the tuple and the object structure that was used to produce it. Such a tuple can then be read by a code generator which produces C++ code that implements classes corresponding to these objects. (This is somewhat analogous to the ROOT `MakeClass` facility.)

Here’s an example of how these classes might be used.

```
1 void sample (TTree* tree)
2 {
3     Long64_t entry = 0;
4     EventInfoObject event (entry);
5     ElectronObject el (entry, "el_");
6
7     event.ReadFrom (tree);
8     el.ReadFrom (tree);
9
10    for (entry = 0; entry < tree->GetEntries(); ++entry) {
11        std::cout << "Event number " << event.EventNumber() << std::endl;
12        for (Int_t i = 0; i < el.n(); ++i)
13            std::cout << " Ele " << i << " pt " << el[i].pt() << std::endl;
14    }
15 }
```

A Python-based package also exists which similarly lets one access the tuple data in an object-oriented fashion. No separate code generator is needed for this case, though, due to the dynamic nature of the Python language.

8. Experience in ATLAS

The tuple framework was in place for the first collision data taken with ATLAS, and was used for early performance studies. Since then, the usage of these tuples has expanded dramatically. They are now used by the majority of ATLAS physics analyses, and have replaced most custom tuple formats.

Tuple making using this toolkit is now integrated into the ATLAS production system. An issue that arose is that making the tuples was taking too much CPU time, up to one CPU second per event. This time was being taken not by the tuple framework, but by other ATLAS tools which were being called from `BlockFiller` tools. A contributing factor to this was that during the rapid growth of the use of tuples, no timing studies were done as new information was added to the tuple. Once profiling studies were done, several bottlenecks were found for which simple fixes sped up tuple making several times. Further improvements are expected as this is studied further.

9. Summary

The tuple framework described here is now an integral part of the ATLAS physics analysis effort, with the majority of physics analyses relying on it. It provides a modular way for physics groups to construct tuples appropriate to their specific analyses while at the same time allowing them to share implementation and validation efforts. It also provides a common set of names and conventions, making it feasible to provide tools that can be useful across many different tuple types. This contributes to the ability of ATLAS to quickly analyze physics data.

Acknowledgments

This work is supported in part by the U.S. Department of Energy under contract DE-AC02-98CH10886 with Brookhaven National Laboratory.

References

- [1] Brun R and Rademakers F 1997 *Phys. Res. A* **389** 81–6 URL <http://root.cern.ch>
- [2] van Gemmeren P and Malon D 2009 *IEEE Int. Conf. on Cluster Computing and Workshops, 2009, New Orleans, USA* pp 1–8
- [3] Duceck G *et al.* (ATLAS) 2005 ATLAS computing technical design report Tech. Rep. CERN-LHCC-2005-022 CERN
- [4] Barrand G *et al.* 2001 *Comp. Phys. Comm.* **140** 45
- [5] The HDF group 2000–10 Hierarchical data format version 5 URL <http://www.hdfgroup.org/HDF5>
- [6] Ehrenfeld W *et al.* (ATLAS) 2011 *J. Phys: Conf. Ser* **331** 032007