Two Proposals

Walter Brown
Marc Paterno
William Tanenbaum
CD-doc-468-v1

Contents

But man acts from judgment,
because by his apprehensive power
he judges that something should be avoided or sought.
— THOMAS AQUINAS

1 Introduction

In a previous document [CD-doc-467], we presented three proposals addressing the issue of a desired "ROOT/COBRA interface." Verbal and email reactions[1] to that document indicated that there was not yet agreement on the requirements or their priorities. Both the main proposal in §3 *and* the alternative proposal of §4.2 received some support.

In this document, we re-present those two proposals, and compare and contrast them with each other and with the existing COBRA framework.[2] The first proposal, which originated from §3 and is herein termed Proposal A, is mostly the same as originally presented, but has been modified to reflect some of the feedback we have received. The

[1] Respondents (in alphabetical order) included: James Branson, Vincenzo Innocente, Norbert Neumeister, Lucia Silvestris, Paris Sphicas, David Stickland, and Stephan Wynhoff.

[2] By COBRA framework we mean the standard CMS batch framework, including the CMS event-data objects defined in ORCA, but used for *analysis* rather than for *reconstruction*.

second proposal, which originated from §4.2 and is herein known as Proposal B, has been substantially augmented and is thus presented in much greater detail than in its previous incarnation. Our goal herein is to present the Proposals with sufficient detail to enable CMS to evaluate their respective individual merits.

2 Desiderata

The document CD-doc-467 enumerated the following desiderata for a ROOT/COBRA plug-in.

1. We want to read existing COBRA data.
2. We want to train people to use the COBRA data model.
3. We want it to be easy to produce distributions from the data.
4. It should be easier to use than the full COBRA framework, but need not be as fast as reading ROOT ntuples.
5. Export COBRA and ORCA high-level objects to ROOT, without exposing the data as ROOT *TTree* instances.
6. We want a plug-in to "make COBRA/ORCA objects readable at the ROOT prompt."

This list contains contributions from several CMS collaborators, and (as previously noted) some of the entries seem to conflict with others. Furthermore, some of the comments we received on CD-doc-467 indicated to us that there was possible disagreement on the meaning of the first three desiderata. In order to help assure we all agree on the meaning of these desiderata, here we explain each of these three in slightly more detail.

We take item 1 to mean that one does not want to employ a *separate* program to transform the COBRA format (perhaps we should say POOL format) data into a format suitable for analysis in ROOT. The reason for this requirement is not entirely clear to us.

We take item 2 to mean that one wishes users of the new software to be learning the skills needed to work in the COBRA framework. There are several skills involved: facility with the C++ programming language, familiarity with the COBRA event-data and reconstruction models, and familiarity with the ORCA classes representing the reconstructed data. Since one does not obtain facility in using things by avoiding them, we surmise this requirement involves *not* shielding the user from all of them. However, in "training" users, it might be useful to do away with some of the complexity of the learning process, perhaps by shielding the user from part of the complexity of the COBRA/ORCA system.

We take item 3 to mean that it should be possible to perform some useful physics analysis tasks (albeit only relatively simple ones) without writing any C++ code. We have *not* assumed that this requirement means that it must be possible to perform *sophisticated* analysis without writing C++ code. We have also *not* assumed that a point-and-click interface is necessary.

3 Overview of the Two Proposals

In this section, we provide an overview of the two proposals. Section 4 describes Proposal A in greater detail, and section 5 is devoted to Proposal B.

Proposal A and Proposal B independently meet many of the same desiderata. Each Proposal provides a tool to be used from within a ROOT session. Each provides access to the CMS event data and the ability to read existing (POOL format) files. Each Proposal allows the user to work in the ROOT interactive environment, and each provides the same easier specification of the input collections. Because what each Proposal provides could *also* be done from a COBRA framework program, we believe it is useful to describe each functionality in terms of such a program.

Proposal A provides a tool that takes POOL files and produces files in a ROOT file format that is more suitable for interactive use. It puts the *RecObj* instances selected by the user into the branches of a *TTree* instance. It allows the user to do so without a need to understand COBRA's mechanisms for accessing reconstructed objects.

To achieve comparable functionality in the current COBRA framework, a user would have to create a module that could be configured (through an `orcarc` file, perhaps) with the names of the *RecObj* classes which were to be written to the created *TTree* output. Such a task seems suitable for batch, rather than interactive, use, because the conversion of large input files will be time-consuming. Such a task seems suitable for interactive use only when the data sample being processed is small. After the *TTree* instances have been created, the user has available all the features of the reconstructed objects subject to any restrictions ROOT might place upon them.

Proposal B provides a tool that allows the execution of arbitrary user code in a COBRA event loop driven from an interactive ROOT session. In this it is very similar to direct use of the COBRA framework. The user would write essentially the same code in each case.

Proposal B requires the user to write C++ code to do even simple tasks. Proposal A does not require the user to write C++ code to do simple tasks; they can be done writing DRAW commands.

It will probably not be possible to perform "complicated" tasks using Proposal A, where "complicated" means any task that requires associating objects in one branch with objects in another using information *other* than the fact that the objects are part of the same event. In particular, some links between objects in the trees will not be usable. Proposal B makes available all the information in the data.

In the existing COBRA framework, users have found it difficult to provide the list of dynamic libraries needed by their jobs. Each Proposal has the same requirement. Ease of use would require the determination of this list be automated. The use of ROOT does not help solve this problem, and probably any solution could be used in COBRA as well as with either Proposal.

Users of the COBRA framework also have complained about the length of the edit/-compile/link/run cycle during analysis. Proposal A avoids this cycle by transforming the data into ROOT's "natural" format, and so enables the user to use ROOT's interactive

facilities for analysis. Proposal B does not alter this cycle. Neither Proposal makes the (potentially time-consuming) step of reading the COBRA-format data faster.

User code developed in the context of Proposal B may be directly transported to the COBRA framework (and thus used in ORCA); the code would be nearly identical.

Finally, we note that Proposal B does not shield the user from the complexities of navigation of the web of reconstructed objects.

4 Proposal A: *RecCollection*s in *TTree*s

This Proposal A would create a facility

- that allows reading COBRA data from within a ROOT session, and
- that would provide automatic reorganization of a subset of the COBRA data into the ROOT format most useful for use from the ROOT prompt.

To this end, we propose the creation of a class, *TCobra*, which has the responsibility of creating ROOT-format data from the COBRA data.[3] This class would coordinate reading of the COBRA-format data, creation, filling, and management of *TTree*s to contain the data, and all associated ROOT "housekeeping" necessary for this to work.

In sections 4.1–4.4 we sketch the important operations of *TCobra*. Section 4.5 shows some uses of the resulting *TTree*, section 4.6 notes some caveats, and section 4.7 mentions additional functionality that could be added in a later release, if such enhancements seem warranted.

4.1 Connecting to CMS Event Data

To create a *TCobra* object, the user specifies the data to be read. Construction of the *TCobra* object does *not* cause the data to be read; reading is delayed until a later step in the use of the object. It is possible to create multiple *TCobra* objects, but it is *not* possible to copy (by copy construction or copy assignment, or cloning) a *TCobra* object. This is disallowed because of the cost of copying the underlying ROOT objects; unintentional copying would likely cause terrible performance problems. If, for some reason, a user wishes to create two identical *TCobra* objects, it is always possible to construct two such objects independently, using the same specification of input data.

```
1 // Specify the data to be read. Only system input collection
2 // specifiers shall be supported in the initial version.
3 TCobra c("/System/aaa/bbb/ccc",
4          "InputFileCatalogURL1␣InputFileCatalogURL2");
```

[3] We recognize that COBRA data files *are* ROOT files. However, these ROOT files are organized in a fashion that is useful for reconstruction, but that is *not* convenient for use from the ROOT prompt. The organization imposed by the transformation done by *TCobra* provides this convenience of use.

4.2 Creating a *TTree*

Each *TCobra* object is limited to creation of a single *TTree* instance. Supporting use of more than a single instance would increase the complexity of the user interface, with no important gain in functionality that we can identify.

The *TTree* instance is associated with a given operating system file and with a *TFile* object, which is used as a backing store. When the *TCobra* object is destroyed (at the end of the ROOT session, if not before), the operating system file remains.

```
// Specify the name of the TTree to be created, and the name of
// the file that shall be used as its backing store.
c.createTree("mytree", "myfile");
```

4.3 Specifying the Objects to be Accessed

The user can specify the objects to be written to the *TTree* instance by calling the member function `createBranch` once for each object to be written. The user specifies:

1. the name of the *RecAlgorithm* whose output is wanted, and
2. the name of the branch to which the output shall be written.

The *RecCollection* produced by the specified *RecAlgorithm* shall be "unwound" into its separate components, which shall be written into a *TClonesArray* on the branch. The call to `createBranch` does not cause reading of the input data.

If the user requests an EDM object which is *not* in the specified data, reconstruction on demand will not be used to provide the requested object. In the first release, only the default version of the algorithms will be used, and **no** ability to do reconstruction on demand will be available. **Please note this behavior is the *opposite* of that which was described in our previous document [CD-doc-467].**

```
// Specify which EDM objects are to be read,
// and the name of the branch to which it shall be written.
c.createBranch("CombinatorialTrackFinder", "ctf");
c.createBranch("GlobalMuonTrackFinder",    "gmtf");
```

The user can list all available branch names and their respective collections, one line per branch, by calling the `listBranches` member function.

```
// List all available collections and their branch names.
c.listBranches(std::cout);
```

4.4 Filling the *TTree*

After the user has specified what branches are to be created, he arranges to fill the branches:

```
// Fill the TTree. Only one of the following should be called.
c.fill();    // to read all events, or
c.fill(100); // to read only 100 events
```

This is the time at which the data is read, and so this step may be very time-consuming.

4.5 Using the Result

After the above calls have been executed, the user has available a ROOT *TTree* instance, with branches containing *TClonesArray*s of the requested EDM objects. The user is able to do whatever ROOT allows to be done with such a *TTree* and its contents.

To help illustrate what will then be possible, we provide several examples. In these examples, we assume `mytree` is a *TTree* containing two branches: the branch *jet* contains jets from some algorithm, and the branch *ele* contains electrons from some algorithm. We also assume that the entries of the *TClonesArray*s have been sorted on `pt`, the transverse momentum of the object.

To obtain a plot of the transverse momentum of *every* jet:

```
root>  mytree.Draw("jet.pt()")
```

To obtain a plot of the transverse momentum of the leading jet in each event:

```
root> mytree.Draw("jet[0].pt()")
```

Note that if some event contained no jets, `TTree::Draw` is smart enough to skip that event.

To obtain a scatter-plot of the transverse momenta of the two leading jets in each event:

```
root> mytree.Draw("jet[0].pt():jet[1].pt()")
```

To obtain a plot of the transverse momentum of the leading electron in each event for which the leading jet had `pt() > 20.0`:

```
root> mytree.Draw("ele.pt()","jet[0].pt()>20.0")
```

For calculations that require looping constructs more complicated than looping over all entries in a *TClonesArray*, the user will generally have to write a CINT macro or C++ code.

4.6 Caveats

Because the time allocated for implementation is short, and because there may be as-yet-unknown technical challenges ahead, we specify some of our assumptions here. If these assumptions turn out to be untrue, it is likely that the project will *not* be possible within the allocated time.

- We assume it is straightforward to create the ROOT dictionaries for all classes for which interactive use is desired.
- We assume there is no problem in having both ROOT and SEAL dictionaries for the same classes in the same program.
- We assume it is straightforward to write the EDM objects into a *TTree*, since these data are currently written in a ROOT format (although that format is different).
- Persistent POOL links between objects are unlikely to be traversable until such time as support for traversing links to objects in *TTree*s is added; adding such support is outside the scope of this project.

We do not know what fraction of the inter-object navigability will be available from within ROOT. The time budget allocated precludes significant modification of the COBRA classes to support use within interactive ROOT.

It is likely that the memory demands of this use of ROOT will be quite large. Code is required for each of the objects to be used; much of this code is required again in the ROOT dictionaries, for interactive use; some code is required a third time in the SEAL dictionaries, for I/O purposes.

Reading data files with this software shall be no faster than when done with the COBRA framework; the same technology is being used. Looping over events in the created *TTree* may be significantly faster. The ROOT files created may be very large. Once the user has selected his choice of branches to be written, no further pruning of data is carried out.

4.7 Possible Later Additions

If user interest dictates, a subsequent version of this software may offer additional functionality. Some enhancements to be considered are:

- more user control over the EDM objects selected, and
- user control over whether reconstruction-on-demand is done.

Another possible addition would allow the user to specify that all available collections be written to the *TTree* instance by calling `createAllBranches`. In this case, default names would be used for the branches. The call to `createAllBranches` would not cause the input data to be read.

```
1 // Specify that all available EDM objects are to be read,
2 // and that they be written to separate branches using
3 // default branch names.
4 c.createAllBranches();
```

4.8 Comparison between Proposal A and the COBRA Framework

We may describe Proposal A as a tool that allows the user to write a ROOT *TTree* from COBRA-format event data. There is very little interactive functionality in such a tool—while the created *TTree* is used interactively, the creation tool itself is not.

Upon reflection, it seems to us that such a tool may be more suitable for a batch environment. A project to create a COBRA framework executable, which the user would configure (perhaps via an `.orcarc` file) to select the *RecObj* classes to be written to the output *TTree*, would have nearly the same utility. Such a project would be significantly easier (and quicker) to complete. If CMS wants a tool that provides an easy means for users to write *RecObj* instances to ROOT *TTree*s, we recommend this alternative rather than Proposal A.

5 Proposal B: The COBRA Framework in ROOT

This Proposal B would create what might be called a ROOT-based analysis framework. By this we mean to provide a class *TCobra*

- that would open and read COBRA-format data files,
- that would provide an event loop, and
- that would allow a user-defined function to be called for each event in the sequence of events.

The user code would be given access to the event, and would use the standard COBRA EDM mechanisms to access elements in the event.

5.1 Connecting to CMS Event Data

During creation of a *TCobra* object, the user specifies the data to be read. Construction of the *TCobra* object does *not* cause the data to be read; reading is delayed until a later step in the use of the object. It is possible to create multiple *TCobra* objects, and it is possible (but seems to be not useful) to copy a *TCobra* object.

```
1 // Specify the data to be read. Only system input collection
2 // specifiers shall be supported in the initial version.
3 TCobra c("/System/aaa/bbb/ccc",
4          "InputFileCatalogURL1␣InputFileCatalogURL2");
```

5.2 The User's Code

We propose that both the following two mechanisms be available to a user in producing his code, and that each user be free to choose whichever one best meets his needs. The first option is function-oriented, and the second is class-oriented.

Under the first option, the user would write a function taking a pointer to an event.[4] There are no arbitrary restrictions placed upon the user code in this function. If the user code calls upon any other facilities of COBRA, it will be his responsibility to ensure that the necessary libraries are loaded.

Under the second option, the user would write a class inheriting from an abstract base class (*TCobraAnalysisModule*) provided as part of Proposal B. The user's class must implement the member function `handleEvent`, taking the same argument as the free function described above. There are no other restrictions placed upon the user's class.

5.3 Running the User's Code

The *TCobra* "event loop" member function `execute` is used to invoke the user's code (whether a free function or class) once per event in the input data.

If `userfunc` is the name of either

1. a free function, as described above, or
2. an instance of the user's class, as described above,

then the following shows how the *TCobra* "event loop" is executed.

```
1 c.execute(userfunc);      // to read all events, or
2 c.execute(userfunc, 100); // to read only 100 events
```

After executing the code above, the user may have histograms, *TTree* instances, or any other artifacts created by the user's code available in the ROOT session.

For example, the user could create and fill the same *TTree* instance as discussed in the exposition of Proposal A, by creating a branch for *RecObj* type in which he is interested, and filling the branches in his user code. It would be a straightforward addition to Proposal B to provide a software component that simplifies this task even further.

5.4 Caveats

It is likely that the memory demands of this use of ROOT will be quite large, albeit slightly less than those for Proposal A. The memory demands should approximate those of the CMS reconstruction program.

[4] The exact type of the pointer needs to be determined by future analysis.

Reading data files with this software shall be no faster than when done with the COBRA framework; the same technology is being used.

5.5 Comparison between Proposal B and the COBRA Framework

We may describe Proposal B as embedding the COBRA framework's event loop in an interactive ROOT session. Users of Proposal B would need knowledge very similar to that needed to make direct use of the COBRA framework. Because the user of Proposal B would need to compile and link analysis code, similar problems arise in the software build phase. Some of the tasks handled by SCRAM would have to be handled directly by the user of Proposal B—or the functionality of SCRAM would have to be duplicated.

The reading of COBRA data may be very time-consuming, especially for large data samples, or for jobs that require additional reconstruction to be done. Such jobs seem less suitable for interactive use than for batch use, especially for those users on systems which limit the CPU time available for interactive jobs.

The main difference between use of Proposal B and direct use of the COBRA framework is that ROOT artifacts (histograms, *TTree* instances, *etc.*) created in the user code of Proposal B would be immediately available in the current ROOT session, while such artifacts written in a COBRA framework program would be written to a file. While this presents some additional convenience for the user, we are concerned that it might also foster poor C++ programming. Because ROOT silently manages the lifetime of many ROOT artifacts, it seems likely that users will become accustomed to this behavior. User code developed in the interactive environment of Proposal B will most likely need careful scrutiny for resource leaks before its inclusion in any widely-used COBRA framework program.

6 Conclusion

If CMS wants an easy-to-use tool for the creation of ROOT *TTree*s filled with ORCA reconstruction objects, we recommend this be achieved by the development of a COBRA/ORCA based application that fills such a *TTree* with user-specified collections of *RecObj* instances. We believe that a project to create such an application will be easier and quicker to complete than Proposal A, while providing the same basic functionality.

If CMS wants a simple-to-use event loop, to invoke the user's code in analysis, then we recommend looking into the underlying reasons for the perceived difficulty of use of the COBRA/ORCA framework.[5] We believe that wrapping the existing event loop to allow its use from within ROOT, while it may have some utility, does not solve the underlying problems.

[5]This may require a large effort, and considerable time.