ProofAna, an analysis framework for HEP data

Bart Butler

Harvard University (ATLAS)

ROOT Workshop, Saas Fee
March 13th, 2013

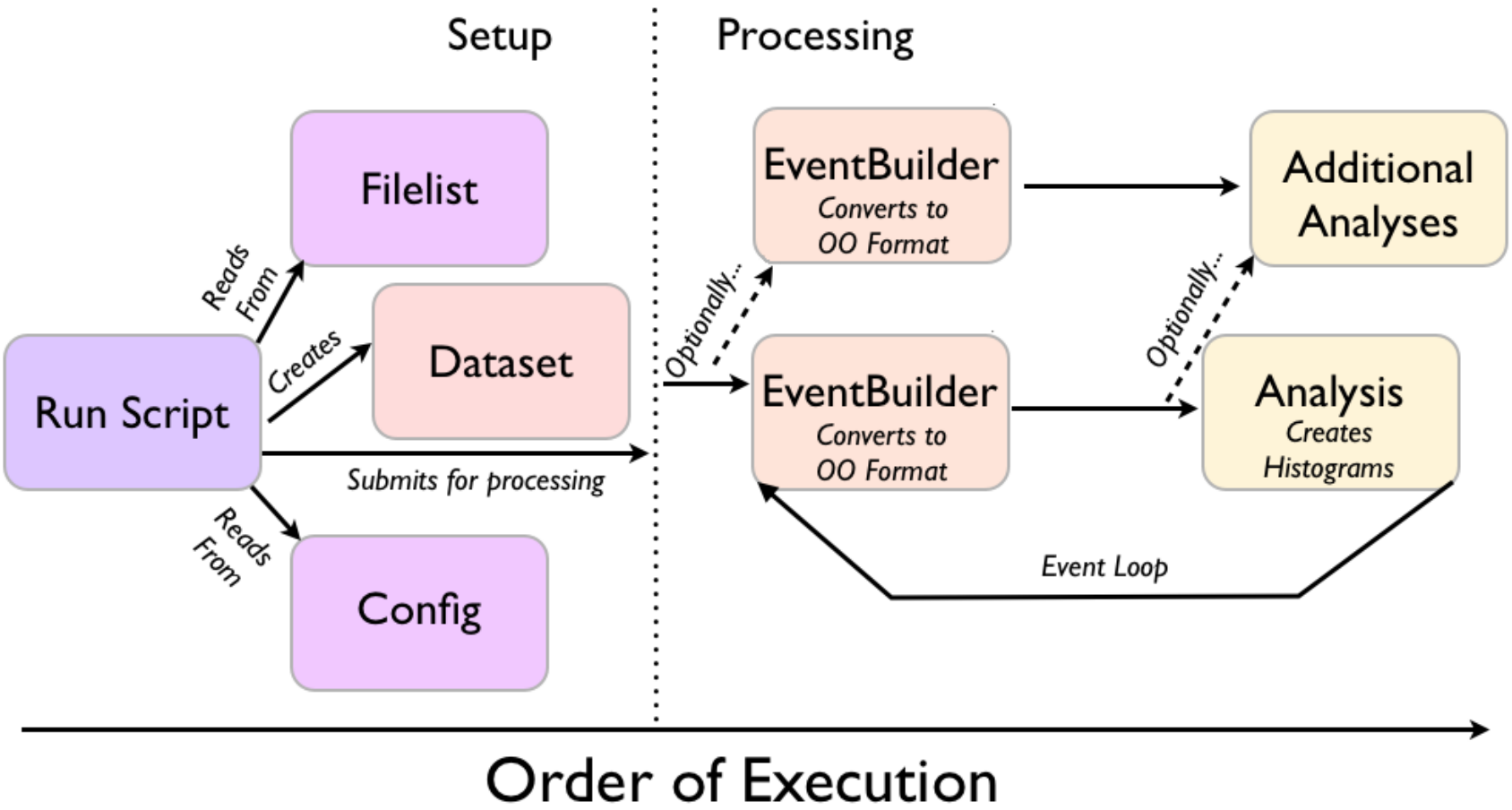
Some History

1. In early 2010, SLAC built the first iteration of its PROOF cluster for ATLAS
   - 49 physical worker cores, 119 TB (usable) xrootd storage
   - Today 325 cores, 755 TB
2. At the time, I was using a recycled D0 framework named FlatAna, which did not support PROOF
3. What originally began as an update to FlatAna became a complete rewrite (not a single line of code was reused) in an attempt to address other shortcomings
4. "ProofAna", the original working title of the project, ended up sticking.
5. Based, of course, on ROOT 5!

Design Goals

- Flexible - If it cannot be reused for many purposes, it is not much of a framework
- Fast - minimal CPU overhead
- **Stupidly-easy to use** - pseudocode-like, minimal learning curve
- Maintenance-friendly
- Engineering controls on ignorance
- Simple interface to useful but otherwise tedious tasks common to physics analysis
- Only "reinvent the wheel" when it provides a tangible benefit

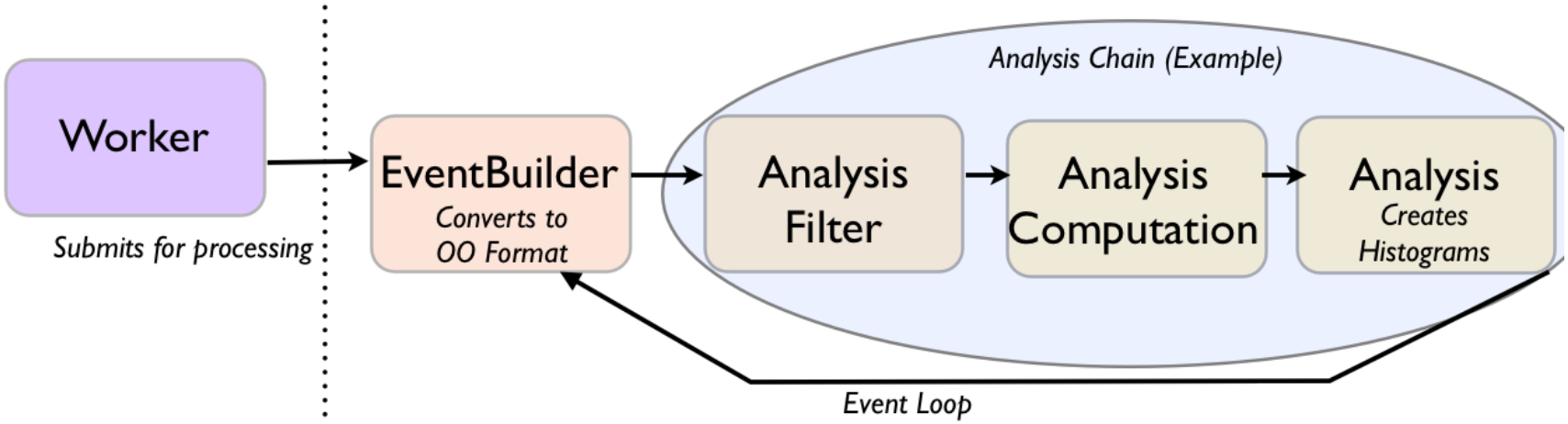
Flexibility I



M. Swiatlowski

Flexibility II

- Everything is "scheduled" in the run script. Analyses are therefore modular, and can be written to be independent of the underlying TTree structure.
- Analyses which do want access to the TTree variables have it.



M. Swiatlowski

Object-Oriented Event Data Model

1. MomentObj
   - Inherits from TObject
   - Key/pair STL map-based container class
   - Stores/retrieves/converts arbitrary "simple type" information: bool, int, float, double, string, etc.
   - Can also store vectors of TObject pointers
   - Fully persistable (ROOT streamers)
2. Particle
   - Public MomentObj
   - TLorentzVector member
3. Point
   - Public MomentObj
   - TVector3 member
4. Event
   - Public MomentObj
   - Additional access functions for object vectors

Example Event 1

"Event" is the top-level container class

- EventNumber (int)
- RunNumber (int)
- electrons (object vector)
  1. Electron 1 (Particle*)
     - nHits (int)
  2. Electron 2 (Particle*)
- muons (object vector)
  1. Muon 1 (Particle*)
     - chi2 (float)
- leptons (object vector)
  1. Electron 1(Particle*)
  2. Muon 1 (Particle*)
  3. Electron 2 (Particle*)

Memory management

- Automatic via reference-counting scheme
- Objects can be "linked" in the Event tree structure as many times as is convenient

Built-in vector sorting by arbitrary moment value, or p_T and E in the case of Particles

Example Event 2

tracks (object vector)

1. Track 1 (Particle*)
   - nHits (int)
2. Track 2 (Particle*)

jets (object vector)

1. Jet 1 (Particle*)
   - tracks (object vector)
     1. Track 2 (Particle*)
   - calibs (object vector)
     1. TLorentzVector 1 (TObject*)
     2. TLorentzVector 2 (TObject*)
     3. TLorentzVector 3 (TObject*)

vtxs (object vector)

1. Point 1(Particle*)
   - tracks (object vector)
     1. Track 2 (Particle*)

Ease of Use

- One class to learn
- As little or as much information as needed
- Relationships between objects in OO way

Maintenance

- Maintenance of MomentObj class is maintenance of the entire event data model

Well that's not going to be fast...

Ease of use/simplicity vs. speed

- Humans like string keys
- String comparisons are slow

Solution

- Transient/internal form of key (especially for comparisons) is an integer
- Persistent form is a string
- String convertible via lookup table - Users can use strings as keys in analysis if they choose, or use the class itself if they care about speed
- Integer-based mapping cache for fast concatenation of keys

So easy a first-year graduate student could do it

```
if( muons("cosmic") ) return true;
BookCutflow("cosmicmuonveto");

if( jets()<2 ) return true;
BookCutflow("2jets");

if( jet(1).p.Pt()<=50.  )  return true;
BookCutflow("2jets50");

if( met()<=25.  )  return true;
BookCutflow("met25");

//b-tagging weights
Weight( Weight()*Float("btagScaleFactor") );

if( jets("btag")!=2 ) return true;
BookCutflow("2btag");
```

Outputs are also easy

The framework provides each analysis a TDirectoryFile in one or more output files. In keeping with minimal reinvention of the wheel, there is no need for wrapping ROOT constructors in custom code:

```
new TH1F(Prefix()+"met",";#slash{E}_{T} [GeV];Events",100,0.,400.);
```

On the other hand, automatic handling of event weights motivates a smart wrapper around Fill functions for histograms and related classes:

```
Fill( "met" , met() );
```

No output object pointer book-keeping $\longrightarrow$ saves coding time

Engineering controls on ignorance I

Example of what this means:

- ROOT I/O is not for beginners
  - MakeClass() and the like on your average ATLAS-style TTree (10,000 branches unslimmed) horribly slow/inefficient.
  - Sometimes even experts are confused about what kind of pointer to pass TTree::SetBranchAddress()

Type-templated TTree branch access

- i.e. Get<int>("branchname")
- Run-time type-checking via template type and SetBranchAddress return value.
- Automatically only load branches that are used and no more
- If third-party code calls SetBranchAddress, this is detected and the Get functions can piggy-back
- Looking forward to TreeReader!

Engineering controls on ignorance II

Another more subtle example:

- EventBuilders are the primary interfaces between TTree data and event data model, and are intended primarily as data-conversion/calibration wrapper classes
- Analyses have access to output files, event weights, and can be chained, but still do have access to the TTree
- Why not just have one base class instead of two? What not make everything an Analysis?

Engineering controls on ignorance III

The separate classes encourage adherence to the analysis model, which results in more efficient physics analysis

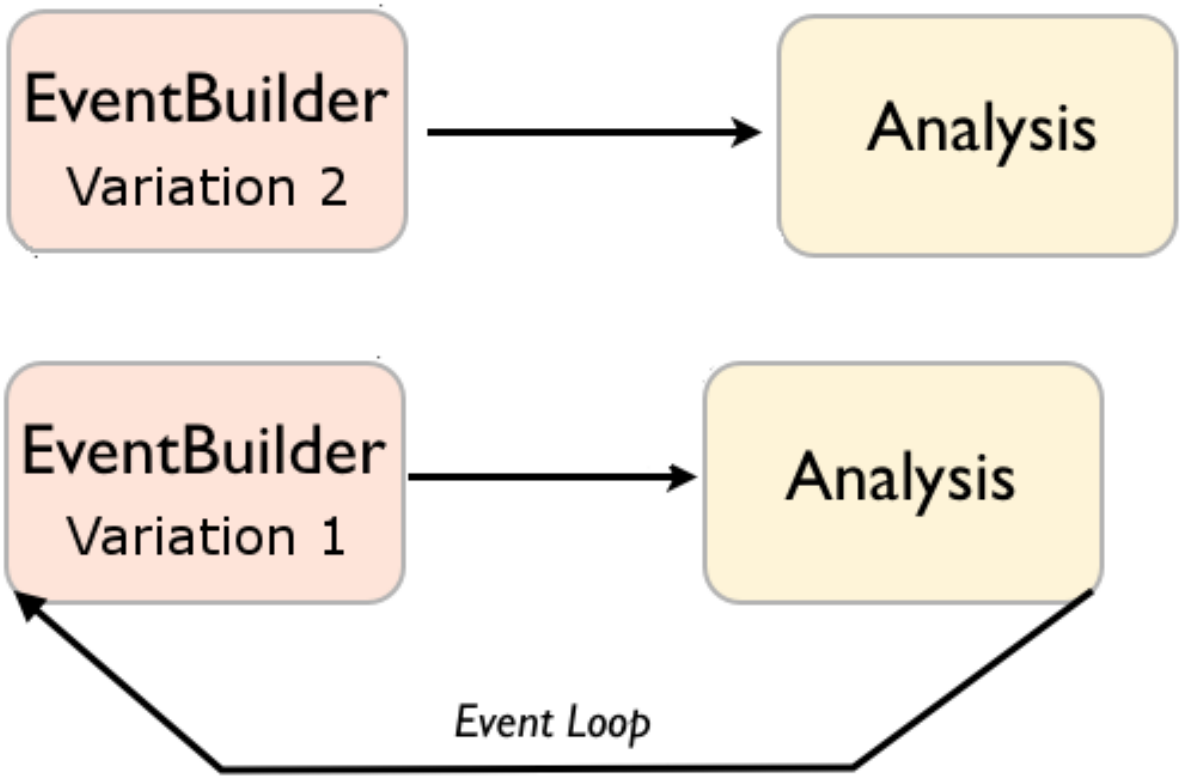
- EventBuilders can be messy, technical, often involve interfacing lots of third-party classes, and are dependent on the TTree structure
- Analyses are often simple (and focused on physics) and TTree-structure-agnostic
- Within the suggested model, people find themselves able to reuse their EventBuilders for other Analyses and vice versa
  - Can hide the details of the "sausage-making" from students and other neophytes
- BUT, as there are many instances where TTree access in Analyses is convenient, compliance with the analysis model is not enforced. Maximum flexibility is therefore preserved.

Tedious tasks made easy (and fast) - Systematics I

I want to run 35 Monte Carlo-based systematic variations

Schedule 35 parallel Analyses, flags to EventBuilder for variations

- Maximal code reuse
- ROOT I/O minimal, as data is only loaded once

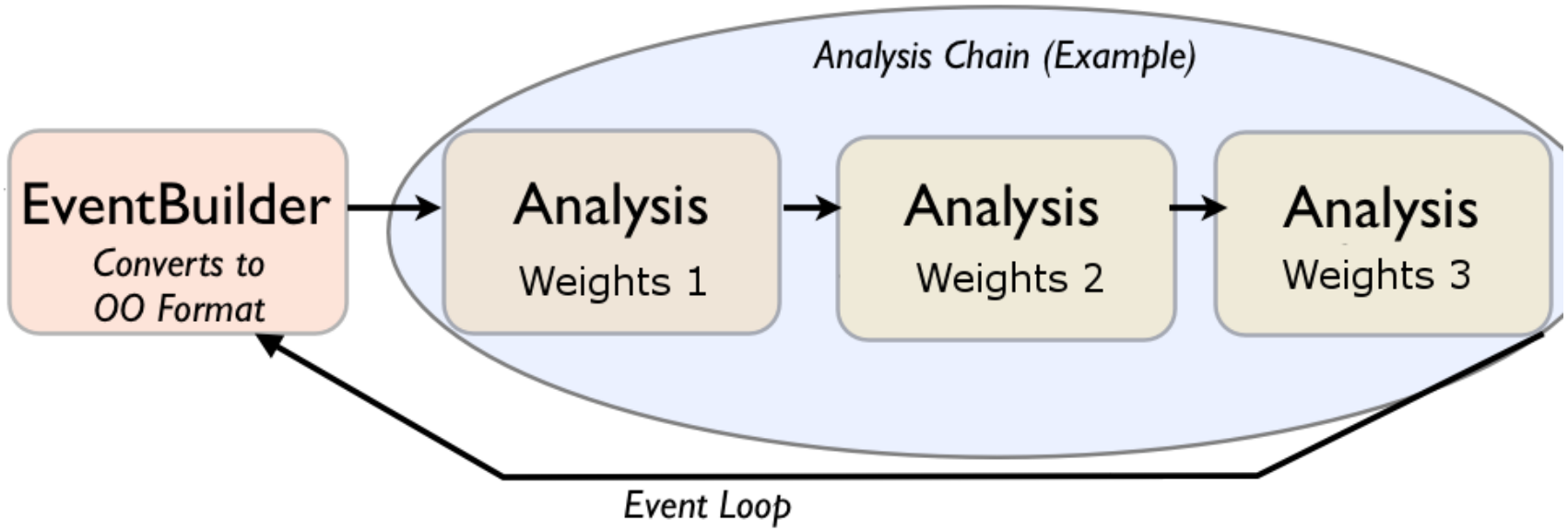


Tedious tasks made easy (and fast) - Systematics II

Some systematic errors just involving reweighting event-by-event

Build the nominal event, feed to an Analysis chain

- Efficient use of CPU (and code)
- Does require some book-keeping – each Analysis must receive a flag to apply the right weight



Tedious tasks made easy (and fast) - TTree Skimming

Reducing the dataset size key to effective analysis

- Somewhat loose skim of the underlying TTree typically done on the Grid
- Want to use "final" kinematics for selections (post-calibration–cannot use raw TTree variables directly)

This is how you do it in a ProofAna analysis:

```
// In job initialization
RegisterTree("susy",false);
// In cut flow
FillTree("susy");
```

If you do this in multiple parallel analyses with the *same name*, you get an OR'd skim:

- 35 $\propto n$ samples or one $\propto \approx 1.5 \times n$ sample is an easy choice

Tedious tasks made easy (and fast) - TTree Slimming

- In ATLAS particularly, a given analysis needs $\mathcal{O}(100)$ branches, and there are $\mathcal{O}(10,000)$ in a centrally-produced TTree
- Need to remove unused branches

This is how you do it in ProofAna:

```
// In job initialization
SlimBranch("branchname")
```

and skim as usual (previous slide).

- There is also a configuration flag which will generate a complete list of all the branches used in the job to aid in putting in these SlimBranch commands.

Tedious tasks made easy (and fast) - OO Skimming

Other use cases:

- Rerunning/rehistogramming very quickly (skip EventBuilder CPU costs)
- Browse final physics objects interactively (CINT, pyROOT) post selections
- Save CPU-intensive new objects that have been generated during your job (i.e. jet-finding)

You could write out a custom TTree or friend tree, or you could:

```
// In job initialization
RegisterTree("sr");
// In cut flow
FillTree("sr");
```

There exists a built-in EventBuilder which handles reading the OO trees and feeding OO events to the Analyses transparently.

Incomplete list of features not discussed previously

- Seamless switching between single-CPU, PROOF-Lite, PROOF cluster, batch queue, and Grid running
  - Change one "connection string"
  - Scripts handle the environment differences
- Single or multiple output files, merging options
  - Can direct events to output based on arbitrary event-based information
- Basic job information/profiling "automatic"
- Simple but intuitive package system, with dependencies
  - Third-party utilities (FastJet, RootCore)
  - User analysis packages
- Very easy build "system"
  - Add Analysis_blah.cxx and header $\rightarrow$ "make" will compile it, generate a dictionary, and allow you to schedule it.
- Growing script/auxiliary class library
  - bash/pyROOT/CINT
  - TH1 with systematics, non-leaky 1D plotting

Contacts and current user base

$\approx$ 30 regular users at multiple ATLAS institutions (SLAC, Harvard, University of Chicago, University of Buenos Aires, and University of Washington, among others)

(Part-time) Developers:

- Bart Butler (Harvard)
- Max Swiatlowski (SLAC)

ProofAna ATLAS e-group (mailing list): atlas-comp-sw-proofana

ATLAS-protected ProofAna TWiki maintained by Max S.

ATLAS-protected (I think?) SVN repositories for the ProofAna framework and ProofAna packages

No objection whatsoever to moving outside of the ATLAS umbrella if there is external interest.