Mu2e-doc-7654-v1

Mu2e's Use of fcl: A Case Study

Rob Kutschke, Andrei Gaponenko

art Users Meeting

June 17, 2016

Outline

- Background and General Comments
- Design goals
- Single points of maintenance
- Two examples
  - 7 instances of one module with only small variants in configuration
  - Presenting groups of modules to the end user as a single unit
- Base plus delta pattern important for grid usage
  - Deltas can be done by appending to or prefixing to the base

Patterns we show here were developed by many members of the art team and many members of the Mu2e collaboration.

Some Background and General Comments

- With the recent improvements fcl is much more powerful than it was 5 years ago,
- It is not a programming language – it is a language for specifying a static configuration
  - We like this model
  - When we need to script things we use bash/perl/python at the discretion of the person doing the work
    - This leads to large volumes of fcl files but SAM is great at managing them.
  - By design we can do everything we need to by prefixing and postfixing a handful of fcl definitions to a base fcl file.
    - We never parse and edit in place

A Common Use Case

- Prepare 10,000 fcl files for 10,000 grid processes:

```
#include "JobConfig/cd3/cosmic/cosmic_s1_general.fcl"

source.firstRun:  1500
source.firstSubRun:  1
source.maxEvents:  1000000
services.user.SeedService.baseSeed:  669411106

// Also set unique output file names – but they don’t fit here.
```

- The physics is all in the included file, which is kept is in the code release and resolved using FHICL_FILE_PATH.
- The other lines are all script generated
  - Ensures unique EventIds and random seeds across the ensemble
- 10,000 final fcl files go into SAM

Parameter Set Validation

- This is a great step forward and we are very happy that it is now available.
  - We have had some painful times without it.
- We have experimented with it and will add it to our most important modules over the summer.
  - Then extend it to all modules at lower priority.
- You need to do the migration from the bottom up.

Design Goals for Mu2e use of fcl

1) Single points of maintenance
2) A .fcl file that runs correctly interactively can be submitted as a grid job without change

- For a little while we got away with the solution "everyone should just remember all the steps needed to do it right".
- We all know that this does not scale well but the problems started immediately.

Single Points of Maintenance

1) Parameters that affect physics may not have default values in code.
   - The default values must be in .fcl files
   - We have not yet achieved this but we are working toward it.
2) When two modules have a common piece of configuration, define it in a prolog and use it in both places
3) When you need to override a few parameters:
   – Use a base+delta style; do not copy and edit.

- All of the above lead to deep configurations
- I believe that this is a language independent statement
  - It would also be true if we used python, json or xml as a configuration language

No Physics Defaults in Code - 1

- For example:

```
double pmin=
  pset.get<double>(“pmin”);        // Recommended

double pmin=
  pset.get<double>(“pmin”, 100.); // Not allowed
```

- Why?
  - A single well defined place to find the default value.
  - Experience has shown that people change the code, forget about the value in the .fcl file and are confused when results do not change as expected.
    - Or worse, they just assume that they did it right and don’t check!

No Physics Defaults in Code - 2

- Downside: this can lead to big parameter sets:
  - The solution is to factorize:

```
label : {
  module_type : MyModule
  userParams {
    a : 1
    b : 2
  }
  expertParams : @local::myModuleExpertDefaults
}
```

- A final comment:
  - Non-physics parameters may have default values in code
  - ie diagnostics level

Dealing with Deep Configurations

- The following options to art are indispensable:

```
--debug-config <file>  Output post-processed configuration to
                       <file> and exit. Equivalent to
                       env ART_DEBUG_CONFIG=<file> art ...
--config-out <file>    Output post-processed configuration to
                       <file> and continue with job.
--annotate             Include configuration parameter source
                       information.
--prefix-annotate      Include configuration parameter source
                       information on line preceding parameter
                       declaration.
```

- The annotate feature is a very welcome new addition.

Example Drawn From Event Mixing

- Configuring 7 instances of one module
  - The module has many configuration parameters – about 15
  - Only two differ from one instance to another

Event Mixing – Generation Step

- We model of 7 sources of background hits
  - Muon Decay in Orbit
  - Protons from muon nuclear capture
  - Neutrons from muon nuclear capture
  - Photons from muon nuclear capture
  - Deuterons from muon nuclear capture
  - Beam flash
  - Muons that stop on material other than our stopping target
- Generate single particle events from each source.
- Pass through G4
- Only write out events that make hits (small fraction of events)
- Save analog precursors to digis

Event Mixing – Mixing Step

- Read in one signal event (optional)
- Run 7 mix-filter modules
  - Each reads in a random number of events from its set of background files
- For each subsystem, the art::Event now contains 7 (or 8) data products containing analog hit precursors.
- Overlay hit precursors on top of each other, apply a model of the electronics and form digis.
- Ready for reconstruction.

Event Mixing – Base Configuration of a Mixing Module

```
BEGIN_PROLOG
mixinFilenames  : @nil
mixerTemplate:  {
   module_type            : MixMCEvents
   fileNames              : @local::mixinFilenames
   // Many lines elided …
   detail  : {
      mean                   : @nil
      genModuleLabel         : “generate”
      g4ModuleLabel          : “detectorFilter”
      g4StatusTag            : “g4run”
      stepInstanceNames      : @local::stepInstanceNames
      // Many lines elided …
   }
}
END_PROLOG
```

- Two parameters have values of @nil
  - I think of these as “pure virtual” parameters

Comments on the previous slide

- Items inside detail : {} are parsed by Mu2e code
- Items outside detail : {} are parsed parsed by *art* code.

Event Mixing – fcl fragment for a mixing job

```
physics : {
  filters : {
    flashMixer    : @local::mixerTemplate
    dioMixer      : @local::mixerTemplate
    // and 5 more
  }
}
# See docdb-6273 for source of mean values.
physics.filters.flashMixer.detail.mean    : 5.11e-5
physics.filters.dioMixer.detail.mean      : 8.37e-6
// and 5 more
```

- Each background set has many files.
- File names are randomized by the machinery that creates fcl files for grid jobs – filenames are injected at that time.

Event Mixing – Another way to write the previous slide

```
physics : {
  filters : {
    flashMixer :{
      @table::mixerTemplate
      mean : 5.11e-5
    }
    dioMixer : {
      @table::mixerTemplate
      mean : 8.37e-6
    }
    // and so on
  }
}
```

- Previous slide was the only way prior to @table.
- For many cases we prefer this style.

Example Drawn from Track Fitting

- How to treat an ensemble of many modules as a single unit for purposes of top level configuration
- This allows changes to be made inside the unit without any need for changes to end user fcl.
  - For example splitting one module into several

Track Finding and Fitting

- In Mu2e tracking finding and fitting is implemented as a sequence of 8 modules plus one or more final fitter modules
- The final fitter comes in 20 flavors:
  - (+, -) * (e, mu, pi, K, p) * (upstream, downstream)
- This started life as a single module that did everything
- Early refactoring steps were painful to end users until we figured out how to present tracking as a single unit to the end user.

Track Fitting prolog 1/3 – Define a base configuration

```
BEGIN_PROLOG

# Basic configuration of the fitter module
TrkRecFit : {
  module_type            : TrkRecFit
  SeedFit                : @local::KalSeedFit
  KalFit                 : @local::KalFinalFit
  KalDiag                : @local::KalDiagDirect
  SeedCollectionLabel    : @nil
  fitparticle            : @nil
  fitdirection           : @nil
}
```

- This is not a runnable configuration.
- Some @locals refer to other @locals, 3 or 4 deep
- END_PROLOG is in 2 pages …

Track Fitting prolog 2/3 – Define complete configurations

```
# downstream going electron
TrkRecFitDownstreameMinus : {
    @table::TrkRecFit
    SeedCollectionLabel  : "PosHelixFinder"
    fitparticle          : @local::Particle.eminus
    fitdirection         : @local::FitDir.downstream
}

# upstream going electrons
TrkRecFitUpstreameMinus : {
    @table::TrkRecFit
    SeedCollectionLabel  : ”NegHelixFinder"
    fitparticle          : @local::Particle.eminus
    fitdirection         : @local::FitDir.upstream
}
// And so on: ( +, - ) * ( e, mu, pi, K, p) * ( up, down )
```

- These are complete configurations and will run.

Track Fitting prolog 3/3 – define units of work

```
Tracking : {
  producers : {

    FSHPreStereo     : @local::FSHPreStereo
    MakeStereoHits   : @local::MakeStereoHits
    // and 5 more modules

    // Up to 20 complete fitter configurations
    TRFDownstreameMinus    : @local::TrkRecFitDownstreameMinus
    TRFUpstreameMinus      : @local::TrkRecFitUpstreameMinus
    TRFDownstreammuMinus   : @local::TrkRecFitDownstreammuMinus
  }
  TPRDownstreameMinus   : [ … ] // The full sequence of modules
  TPRDownstreammuMinus : [ … ] // The full sequence of modules
  // and so on
}
END_PROLOG
```

Top level fcl file that uses Track Fitting

```
physics : {
  producers : {
    @table::Tracking.producers
  }

  tpath : [ @sequence::Tracking.TPRDownstreameMinus,
            @sequence::Tracking.TPRDownstreammuMinus  ]
  trigger_paths  : [ tpath ]
}
END_PROLOG
```

- This says very clearly what it does: it looks for two types of tracks, downstream going e- and mu-.
  - Goal is to make fcl talk physics, not fcl
- Major refactoring of the tracking modules took place behind the scenes with zero changes to end user fcl files.

It’s still not quite perfect

- If we define all 20 possible fitter modules in the Tracking.producers table but don’t use them all, then we get some warning messages at job start up
  - “Module is not used in any path”
- In the trigger path, the 7 modules that precede the final fitter module appear twice.
  - This is OK since art automatically identifies repeated modules and runs only on the first encounter. It remembers the result on subsequent encounters.
  - The wasted time is trivial on the scale of the whole job

Features to Enable grid processing

- The example originated in EventMixing but it would take too long to give the full story
  - So this is stripped down.

A fragment that works interactively for testing: myjob.fcl

```
BEGIN_PROLOG
 mixinFilenames  = [ “/mu2e/data/users/kutschke/test.art”  ]
END_PROLOG

physics  : {
  producers  : {
    a : {
      @table::A
      inputFile  : @local::mixinFilenames
    }
    b : {
      @table::B
      inputFile  : @local::mixinFilenames
    }
  }
}
```

Statement of the Problem

- I want to run the myjob.fcl interactively for testing and then run exactly the same myjob.fcl on the grid
- In each grid process, I want my grid script to replace the value of the PROLOG parameter mixinFilenames with a different value. The value changes from one process to the next.
- I do not want to parse the file to find everywhere that mixinFilenames is used and change those – I want to just change mixinFilenames and have that change propagate via @local::
- I do not want to edit the file so that the interactive fcl and grid fcl are different.

Attempt 1: This fails

```
#include “mjob.fcl”

mixinFilenames  : [“/mu2e/data/users/kutschke/differentFile.art”]
```

- This fails because mixinFilenames still had it’s original value at the time that the two @local:: were evaluated.

Attempt 2: This fails too

```
mixinFilenames:[“/mu2e/data/users/kutschke/differentFile.art”]

#include “myjob.fcl”
```

- This fails because the definition of mixinFilenames inside myjob.fcl overrides the value at the top of the file. This happens before the @local:: are evaluated.

Attempt 3: This works!

```
mixinFilenames @protect_ignore:   [
“/mu2e/data/users/kutschke/differentFile.art”  ]

#include “myjob.fcl”
```

- This works because the definition of mixinFilenames inside myjob.fcl is silently ignored.
- This the use case that motivated the creation of @protect_ignore.

Thanks to the *art* team!

- We appreciate the help we get from the art team whether we come to them with bug reports or just to ask advice.
  - We usually get a fast turn around and the problem is solved.
- We appreciate the time that they take to ask enough questions so that they can develop a good understanding of what we really need
  - Sometimes we learn that it is very different from what we first thought that we needed!