LHCb future Hands On Tutorial

Sébastien Ponce
sebastien.ponce@cern.ch

CERN

January 23rd 2017

Foreword

What this course is not

- An overall presentation of Gaudi
- An introduction to LHCb software
- A C^{++}course (see here for that)

What it tries to be

- a practical tutorial to use the new LHCb software framework
- a set of examples of good practices
- an help for your future hackathon work
- all examples are extracted from real LHCb code

Outline

Converting to functional

1. Converting to functional
   - TES interaction
   - Algorithm Declaration
   - Constructor
   - operator()
2. Modernizing your code
3. Common complications

Check list for converting to functional

- identify TES interactions
- deduce functional algorithm to use
- modify Algorithm declaration
- modify constructor/destructor
- convert execute into operator(...)

TES interaction

1. Converting to functional
   - TES interaction
   - Algorithm Declaration
   - Constructor
   - operator()

Analyze TES interaction

What to do

- find out what is the input data
- find out what is the output data

How to do

- code inspection
- look/grep for *get*, *put*, *getIfExists*

Example

PrMatchNN.cpp

```
egrep 'get|put' Pr/PrAlgorithms/src/PrMatchNN.cpp
```

```
declareProperty("VeloInput", m_veloLocation=LHC...
declareProperty("SeedInput", m_seedLocation=LHC...
declareProperty("MatchOutput", m_matchLocation=...
  put(matchs, m_matchLocation);
LHCb::Tracks* velos =
  getIfExists<LHCb::Tracks>(m_veloLocation);
LHCb::Tracks* seeds =
  getIfExists<LHCb::Tracks>(m_seedLocation);
```

Example

PrMatchNN.cpp

```
egrep 'get|put' Pr/PrAlgorithms/src/PrMatchNN.cpp
declareProperty("VeloInput", m_veloLocation=LHC...
declareProperty("SeedInput", m_seedLocation=LHC...
declareProperty("MatchOutput", m_matchLocation=...
  put(matchs, m_matchLocation);
LHCb::Tracks* velos =
  getIfExists<LHCb::Tracks>(m_veloLocation);
LHCb::Tracks* seeds =
  getIfExists<LHCb::Tracks>(m_seedLocation);
```

PrMatchNN data flow

Velo Tracks + Seed Tracks → Ouput Tracks

Deduce functional algorithm to be used

Possible inputs

- 0 : no input, pure producer
- 1-n : one or several independent inputs. This number and the type of each input must be known at compile time
- vect : any number of inputs (not know at compile time), all of the same type, aka a vector of inputs of the same type

Possible outputs

- 0 : no output, pure consumer
- 1-n : one or several independent outputs. This number and the type of each output must be known at compile time
- vect : any number of outputs (not know at compile time), all of the same type, aka a vector of outputs of the same type
- boolean : an additional boolean output, saying whether we should carry on or stop the processing, aka a filter

Helper table to choose your functional Algorithm

Out \ In	0	1 - n	vect
0		Consumer	
1	Producer	Transformer	Merging Transformer
n		MultiTransformer	
vect		SplittingTransformer	
boolean		FilterPredicate	
boolean + 1-n		MultiTransformerFilter	

Algorithm Declaration

1. Converting to functional
   - TES interaction
   - Algorithm Declaration
   - Constructor
   - operator()

Class declaration new version

Changes

- change inheritance from Algorithm to your functional algorithm base class
- template the functional algorithm with the "signature" of the algorithm, that is its data flow

Examples

in case of Velo Tracks + Seed Tracks → Output Tracks

```
Gaudi::Functional::Transformer
    <LHCb::Tracks(const LHCb::Tracks&,
                  const LHCb::Tracks&)>
```

in case of Tracks → Vector of Tracks

```
Gaudi::Functional::SplittingTransformer
  <std::vector<LHCb::Tracks>
      (const LHCb::Tracks&)>
```

Class declaration new version

More changes

- all inputs are now using const references
- all declarations of members storing locations of input/output can be dropped (automatic retrieval from TES)
- equivalent properties are automatically defined by the functional framework

Examples

Member declaration :

```
std::string m_veloLocation;
```

Used in constructor as :

```
declareProperty("VeloInput", m_veloLocation=...
```

How it looks like for PrMatchNN

Old

```
#include "GaudiAlg/GaudiAlgorithm.h"
class PrMatchNN : public GaudiAlgorithm {
  ...
  std::string m_veloLocation;
  std::string m_seedLocation;
  std::string m_matchLocation;
```

New

```
#include "GaudiAlg/Transformer"
class PrMatchNN :
  public Gaudi::Functional::Transformer
  <LHCb::Tracks(const LHCb::Tracks&,
                const LHCb::Tracks&)> {
  ...
```

Constructor

1. Converting to functional
   - TES interaction
   - Algorithm Declaration
   - Constructor
   - operator()

Constructor new version

Changes

- initialize functional algorithm rather than Algorithm
- New arguments are needed :
  - input location / list of input locations in TES
  - output / list of inputs locations in TES
- Locations are given using KeyValue objects
- This creates the adequate property in the back
  - so old declareProperty lines can be dropped
- For each location, a default might be given

Example of Location

```
KeyValue{"VeloInput", LHCb::TrackLocation::Velo}
```

How it looks like for PrMatchNN

Old

```
PrMatchNN::PrMatchNN(const std::string& name,
                     ISvcLocator* pSvcLocator) :
  GaudiAlgorithm(name, pSvcLocator), ... {
  declareProperty("VeloInput", m_veloLocation=...
  declareProperty("SeedInput", m_seedLocation=...
  declareProperty("MatchOutput", m_matchLocation=...
```

New

```
PrMatchNN::PrMatchNN(const std::string& name,
                     ISvcLocator* pSvcLocator) :
Transformer(name, pSvcLocator,
  {KeyValue{"VeloInput", LHCb::TrackLocation::Velo},
   KeyValue{"SeedInput", LHCb::TrackLocation::Seed}},
  KeyValue{"MatchOutput", LHCb::TrackLocation::Match}),
... {
```

operator()

1. Converting to functional
   - TES interaction
   - Algorithm Declaration
   - Constructor
   - operator()

Convert execute into operator(...)

Declaration of operator(...)

- signature matching the data flow and using const references
- of the form

```
Output operator()
  (const Input&, ...) const override;
```

- note the const and override keywords

For PrMatchNN

```
LHCb::Tracks operator()
  (const LHCb::Tracks& velos,
   const LHCb::Tracks& seeds) const override;
```

Convert execute into operator(...)

Implementation of operator(...)

- keep execute's code but
  - drop all interactions with TES (get/put)
  - replace memory allocation of the output by simple creation on the stack
  - thus change accordingly '->' into '.', drop some '*'
  - return output rather than a StatusCode
  - throw an exception when you were returning a bad StatusCode

How it looks like on PrMatchNN

Note : simplified, checks have been removed

Code of operator(...)

```
LHCb::Tracks* matchs = new LHCb::Tracks;


put(matchs, m_matchLocation);
LHCb::Tracks* velos =
    getIfExists<LHCb::Tracks>(m_veloLocation);
LHCb::Tracks* seeds =
    getIfExists<LHCb::Tracks>(m_seedLocation);
StatusCode sc = m_matchTool->match
  (*velos, *seeds, *matchs);


return sc;
```

How it looks like on PrMatchNN

Drop interaction with TES

Code of operator(...)

```
LHCb::Tracks* matchs = new LHCb::Tracks;


put(matchs, m_matchLocation);
LHCb::Tracks* velos =
    getIfExists<LHCb::Tracks>(m_veloLocation);
LHCb::Tracks* seeds =
    getIfExists<LHCb::Tracks>(m_seedLocation);

StatusCode sc = m_matchTool->match
  (*velos, *seeds, *matchs);


return sc;
```

How it looks like on PrMatchNN

Replace memory allocation

Code of operator(...)

```
~~LHCb::Tracks* matchs = new LHCb::Tracks;~~

LHCb::Tracks matchs;

~~put(matchs, m_matchLocation);~~
~~LHCb::Tracks* velos =~~
    ~~getIfExists<LHCb::Tracks>(m_veloLocation);~~
~~LHCb::Tracks* seeds =~~
    ~~getIfExists<LHCb::Tracks>(m_seedLocation);~~

StatusCode sc = m_matchTool->match
  (*velos, *seeds, *matchs);


return sc;
```

How it looks like on PrMatchNN

Adapt code

Code of operator(...)

```
LHCb::Tracks* matchs = new LHCb::Tracks;

LHCb::Tracks matchs;

put(matchs, m_matchLocation);
LHCb::Tracks* velos =
    getIfExists<LHCb::Tracks>(m_veloLocation);
LHCb::Tracks* seeds =
    getIfExists<LHCb::Tracks>(m_seedLocation);

StatusCode sc = m_matchTool->match
  (*velos, *seeds, *matchs);


return sc;
```

How it looks like on PrMatchNN

return object

Code of operator(...)

```
LHCb::Tracks* matchs = new LHCb::Tracks;

LHCb::Tracks matchs;

put(matchs, m_matchLocation);
LHCb::Tracks* velos =
    getIfExists<LHCb::Tracks>(m_veloLocation);
LHCb::Tracks* seeds =
    getIfExists<LHCb::Tracks>(m_seedLocation);

StatusCode sc = m_matchTool->match
  (*velos, *seeds, *matchs);

return matchs;

return sc;
```

How it looks like on PrMatchNN

Final code is free of all boiler plate

Code of operator(...)

```
LHCb::Tracks matchs;
m_matchTool->match (velos, seeds, matchs);
return matchs;
```

You're done !

Your code is using the functional framework.

You're done !

Your code is using the functional framework.

Benefits you will get

- much simple and more readable code
- faster code
- parallelisation of your algorithm with others
- ability to run your algorithm on several events concurrently
- checks for thread safety thanks to constness

So it may not compile straight...

Modernizing your code

Modernization

Main Idea

Take benefit of the code review triggered by functional changes

- for simplifing property usage
- for using C++11/14/17 goodies
- for checking the thread safety of the code

Properties

2. Modernizing your code
   - Properties
   - Counters
   - $C^{++}11$
   - Check thread safety

Properties have improved

What changed

- Gaudi::Property class was introduced
- This is the new way of declaring properties
- simple declaration is enough
- declareProperty is (almost) obsolete

Properties have improved

What changed

- Gaudi::Property class was introduced
- This is the new way of declaring properties
- simple declaration is enough
- declareProperty is (almost) obsolete

Caveat

- not completely backward compatible
- some code may not compile
- but fixes are trivial
- some code may compile but change behavior !!
- fixes also trivial once identified

Practically

Old .h file

```
bool m_skipFailedFitAtInput;
/// Max chi2 per track
double m_maxChi2DoF;
```

Old .cpp file (in constructor)

```
declareProperty("SkipFailedFitAtInput",
                m_skipFailedFitAtInput=true);
declareProperty("MaxChi2DoF",
                m_maxChi2DoF=9999);
```

New .h file

```
Gaudi::Property<bool> m_skipFailedFitAtInput
  { this, "SkipFailedFitAtInput", true };
Gaudi::Property<double> m_maxChi2DoF
  {this, "MaxChi2DoF", 9999, "Max chi2 per track"};
```

Non backward compatibility

Almost backward compatible

- `Property<T>` will auto convert to `T`
- most operators are supported, e.g. += for `Property<int>`

Compilation failure

```
Gaudi::Property<T*> m_pointer
    {this, "P", mypointer};
m_pointer->foo();
```

Solution

```
Gaudi::Property<T*> m_pointer
    {this, "P", mypointer};
m_pointer.value()->foo();
```

More tricky non backward compatibility

No compilation error this time

```
Gaudi::Property<std::string> m_reason
  {this, "Reason", "", "Reason"};
info() << "Abort : " << m_reason << endmsg;
```

Will print the property (Name : Value) rather than the value only

Same solution

```
Gaudi::Property<std::string> m_reason
  {this, "Reason", "", "Reason"};
info() << "Abort : " << m_reason.value() << endmsg;
```

Counters

2 Modernizing your code
- Properties
- Counters
- $C^{++}11$
- Check thread safety

Switch to new counters

They are available in latest Gaudi

- they are faster and easier to use
- they allow thread buffering
- all details in separate presentation

https://twiki.cern.ch/twiki/bin/view/LHCb/LHCbSoftwareTutorials

C++11

2. Modernizing your code
   - Properties
   - Counters
   - C++11
   - Check thread safety

Use member initialization

To be changed

- move defaults values for members from constructor to declaration

Gain

- simplifies the constructor
- fixes automatically the initializations you forgot

Member initialization example

.h Before

```
IHitExpectation* m_itExpectation;
IHitExpectation* m_otExpectation;
```

.cpp Before

```
TrackComputeExpectedHits(...) :
  GaudiAlgorithm(name, pSvcLocator),
  m_itExpectation(0), m_otExpectation(0) {}
```

.h After

```
IHitExpectation* m_itExpectation = nullptr;
IHitExpectation* m_otExpectation = nullptr;
```

.cpp After

```
using GaudiAlgorithm::GaudiAlgorithm;
```

Drop uneeded constructors/destructors

To be changed

- drop unnecessary empty destructors
  - often the case in Algorithms
- replace empty constructors using `using`

Example of constructor drop

Before

```
std::string m_inputLocation;
VertexListFilter::VertexListFilter
  (const std::string& name,
   ISvcLocator* pSvcLocator) :
  GaudiAlgorithm(name, pSvcLocator) {
  declareProperty("InputLocation",
                  m_inputLocation=...);
 }
```

After

```
Gaudi::Property<std::string> m_inputLocation
  {this, "InputLocation", ... } ;
using GaudiAlgorithm::GaudiAlgorithm;
```

Use auto and range-based loops

Before

```
const std::vector<LHCb::LHCbID>& ids =
  aTrack->lhcbIDs();
for (std::vector<LHCb::LHCbID>::const_iterator it =
     ids.begin(); it != ids.end(); ++it){
  if ((it->isOT() == true) ||
      (it->isIT() == true))
    tSeed->addToLhcbIDs(*it);
}
```

After

```
for (const auto& id : aTrack->lhcbIDs()) {
  if (id.isOT() || id.isIT())
    tSeed->addToLhcbIDs(id);
}
```

Use lambdas with the STL

Before

```
return std::count_if
  (lhcbids.begin(),
   lhcbids.end() ,
   boost::lambda::bind(m_pmf,
                       boost::lambda::_1));
```

After

```
return std::count_if
  (lhcbids.begin(),
   lhcbids.end() ,
   [&](const LHCb::LHCbID& id) {
     return (id.*m_pmf)();
   });
```

Check thread safety

2. Modernizing your code
   - Properties
   - Counters
   - C++11
   - Check thread safety

the ways to check thread safety

The theory

- let the compiler do it via constness
  - multiple read-only accesses are safe
  - but a concurrent write raises issues
- so just use the functional framework !
- check you do not use incidents

the ways to check thread safety

The theory

- let the compiler do it via constness
  - multiple read-only accesses are safe
  - but a concurrent write raises issues
- so just use the functional framework !
- check you do not use incidents

The practice

There are ways to work around the compiler checks

- const_cast
- mutables

the ways to check thread safety

The theory

- let the compiler do it via constness
  - multiple read-only accesses are safe
  - but a concurrent write raises issues
- so just use the functional framework !
- check you do not use incidents

The practice

There are ways to work around the compiler checks

- const_cast
- mutables

and LHCb uses them extensively !!!

Incidents

Have to be dropped

- they are incompatible with multithreading
- thus they are not even raised in GaudiHive
- so code relying on them will misbehave

Incidents

Have to be dropped

- they are incompatible with multithreading
- thus they are not even raised in GaudiHive
- so code relying on them will misbehave

Looking for incidents

Incident handler :

```
void handle(const Incident&) override final;
```

Registration of the handler in constructor :

```
incSvc()->addListener(this, IncidentType::EndEvent);
```

How to remove incidents

The easy (and standard) case

- incident only needed to reset a cache at event start
- the cache itself needs to be removed to be thread safe
- so refer to further topic on dealing with cache removal

The general (seldom) case

- incident is a communication mean between 2 entities
- the information passed needs to be passed through regular channels
  - function arguments
  - object in TES
- one needs to analyse the information flow of the application

const_casts

What it is

A way to cast a const type to non-const

```
void foo(const T& const_param) {
  T& param = const_cast<T&>(const_param);
  param->setField(...);
}
```

const_casts

What it is : a killer for thread safety !

A way to cast a const type to non-const

```
void foo(const T& const_param) {
  T& param = const_cast<T&>(const_param);
  param->setField(...);
}
```

const_casts

What it is : a killer for thread safety !

A way to cast a const type to non-const

```
void foo(const T& const_param) {
  T& param = const_cast<T&>(const_param);
  param->setField(...);
}
```

Why removing them ?

- they hide problems by preventing compiler checks
- very often they are introduced with that very purpose !
- they are just evil !

const_casts

What it is : a killer for thread safety !

A way to cast a const type to non-const

```
void foo(const T& const_param) {
  T& param = const_cast<T&>(const_param);
  param->setField(...);
}
```

Why removing them ?

- they hide problems by preventing compiler checks
- very often they are introduced with that very purpose !
- they are just evil !

Solution

Just drop them (and deal with the uncovered issues)

mutables

What it is

Marks a class member as not targeted by method constness

Useful for implementing caches

```
struct C {
  mutable int last_used_value;
  void foo(int a) const {
    last_used_value = a;
    ...
  }
}
```

mutables

What it is : another killer for thread safety !

Marks a class member as not targeted by method constness

Useful for implementing caches

```
struct C {
  mutable int last_used_value;
  void foo(int a) const {
    last_used_value = a;
    ...
  }
}
```

mutables

What it is : another killer for thread safety !

Marks a class member as not targeted by method constness

Useful for implementing caches

```
struct C {
  mutable int last_used_value;
  void foo(int a) const {
    last_used_value = a;
    ...
  }
}
```

Why removing them ?

- same reasons as for const_cast

Removing mutables

The easy (and standard) case

- the mutable is caching some information between calls
- see further topic on dealing with cache removal

The general case (seldom, mostly framework)

- make sure using mutable is the best way
- protect it with atomics or mutexes
  - mutex will have to be mutable too

Common complications

1. Converting to functional
2. Modernizing your code
3. Common complications
   - Multiple output example
   - Specify Algo base class
   - Tricky TES accesses
   - States and caches

Purpose of this part

- go through a number of standard problems encountered when converting code along the lines discussed so far
- give ideas of solutions/work arounds based on my experience

Multiple output example

3 Common complications
- Multiple output example
- Specify Algo base class
- Tricky TES accesses
- States and caches

Syntax for functional Algos with multiple outputs

Declaration

The return type uses an std::tuple

```
typedef Gaudi::Functional::MultiTransformerFilter
  <std::tuple<LHCb::RecVertices,
              LHCb::PrimaryVertices>
  (const LHCb::Tracks&)>
  PatPV3DBaseClass;
class PatPV3D : public PatPV3DBaseClass {
```

Here a typedef is used for readibility

Syntax for functional Algos with multiple outputs

Constructor

As for mutiple inputs, the locations are given via a “list” expression, using braces

```
PatPV3D::PatPV3D(const std::string& name,
                 ISvcLocator* pSvcLocator) :
  MultiTransformerFilter(name , pSvcLocator,
    KeyValue{"InputTracks", ...},
    {KeyValue("OutputVerticesName", ...),
     KeyValue("PrimaryVertexLocation", "")})
    {
```

Syntax for functional Algos with multiple outputs

operator()

- uses a tuple that needs to be build on return
  - std:move may be useful then
- for a filter, the first item in the returned tuple is a boolean

```
std::tuple<bool, LHCb::RecVertices,
                 LHCb::PrimaryVertices>
PatPV3D::operator()(const LHCb::Tracks& ...) const {
  ...
  LHCb::RecVertices outputRecVertices;
  LHCb::PrimaryVertices outputPrimaryVertices;
  ...
  return std::make_tuple
    (true, std::move(outputRecVertices),
           std::move(outputPrimaryVertices));
}
```

Specify Algo base class

3. Common complications
   - Multiple output example
   - Specify Algo base class
   - Tricky TES accesses
   - States and caches

When Algorithm has specific base class

It can be specified using "Traits"

passed as second template argument of the functional algorithm base class

Old code

```
class VeloIPResolutionMonitor :
  public GaudiHistoAlg {
```

Old code

```
class VeloIPResolutionMonitor :
  public Gaudi::Functional::Consumer
  <void(const LHCb::Tracks&, ...),
   Gaudi::Functional::Traits::BaseClass_t
   <GaudiHistoAlg>> {
```

Tricky TES accesses

3. Common complications
   - Multiple output example
   - Specify Algo base class
   - Tricky TES accesses
   - States and caches

TES access within a tool

The problem

- many tools have hidden TES accesses
- they are very often cached
- const_cast or mutables are used to hide it from const checking

TES access within a tool

The problem

- many tools have hidden TES accesses
- they are very often cached
- const_cast or mutables are used to hide it from const checking

Even worse

- suppose no const_cast and no mutable
- and a non-const tool method
- you can still call it from a const method of your Algorithm
    - in other words, the compiler check does not work for tools

Example code

The tool

```
class PVOfflineTool :
  public extends<GaudiTool, IPVOfflineTool> {
  StatusCode reconstructMultiPV
    (const LHCb::Tracks&,
     std::vector<LHCb::RecVertex>&) override;
```

The algorithm

```
class PatPV3D : public PatPV3DBaseClass {
  IPVOfflineTool* m_pvsfit = nullptr;
}
std::tuple<...> PatPV3D::operator()
  (const LHCb::Tracks& inputTracks) const {
  m_pvsfit->reconstructMultiPV(inputTracks, rvts);
```

Example code

The tool

```
class PVOfflineTool :
  public extends<GaudiTool, IPVOfflineTool> {
  StatusCode reconstructMultiPV
    (const LHCb::Tracks&,
     std::vector<LHCb::RecVertex>&) override;
```

The algorithm

```
class PatPV3D : public PatPV3DBaseClass {
  IPVOfflineTool* m_pvsfit = nullptr;
}
std::tuple<...> PatPV3D::operator()
  (const LHCb::Tracks& inputTracks) const {
  m_pvsfit->reconstructMultiPV(inputTracks, rvts);
```

We have a const pointer to non-const tool !

How to sort out tools

Main ideas

- for dealing with caches, see next topic
- for hidden TES access to given data
  - extract access to the algorithms needing that data
  - pass a const reference to data as an extra parameter to tool's methods using it
- for const checking
  - use ToolHandle that will make sure constness is respected
  - or check manually constness of all methods of tools you call from the operator() (recursively)

Example of extracting TES access from tool

Before

```
StatusCode Velo::VeloIPResolutionMonitor::execute() {
  m_pvtool->reDoSinglePV(...);
  ...
}
StatusCode PVOfflineTool::reDoSinglePV(...) {
  auto rtracks = readTracks();
  ...
}
std::vector<const LHCb::Track*>
PVOfflineTool::readTracks() const {
  LHCb::Tracks* stracks = get<LHCb::Tracks>(trName);
  ...
}
```

Example of extracting TES access from tool

After

```
void Velo::VeloIPResolutionMonitor::operator()
  (const LHCb::Tracks& tracks, ...) const {
  m_pvtool->reDoSinglePV(tracks, ...)
  ...
}
StatusCode PVOfflineTool::reDoSinglePV
  (const LHCb::Tracks& inputTracks, ...){
  auto rtracks = readTracks(inputTracks);
  ...
}
std::vector<const LHCb::Track*>
  PVOfflineTool::readTracks
  (const LHCb::Tracks& inputTracks) const {
  ...
}
```

States and caches

3 Common complications
- Multiple output example
- Specify Algo base class
- Tricky TES accesses
- States and caches

Dealing with states/caches while being thread safe

Caveat

- I'll only talk of per event state/cache
- these are the ones impacted by the new framework
- more generic state/cache will need other solutions
    - atomics, mutexes, architecture review

Main ideas (same as for TES access)

- extract the state/cache to the algorithm
    - by having a "createCache" method in the tool
- pass the state as (non const) reference argument to methods dealing with it

Example of extracting state from tool

Tool before

```
class PrVeloUTTool :
public extends<GaudiTool, ITracksFromTrack> {
  mutable std::array<PrUTHits,8> m_hitsLayers;
  mutable std::array<PrUTHits,4> m_allHits;
  ...
  StatusCode tracksFromTrack(...) const {
    ...
    for(auto& ah : m_allHits) ah.clear();
```

Usage before

```
m_veloUTTool = tool<ITracksFromTrack>(...);
for(const auto& veloTr: inputTracks) {
  m_veloUTTool->tracksFromTrack(...);
  ...
}
```

Example of extracting state from tool

After - State definition

```
struct PrVeloUTEventState {
  std::array<PrUTHits,8> m_hitsLayers;
  std::array<PrUTHits,4> m_allHits;
};
```

Example of extracting state from tool

After - Tool

```
class PrVeloUTTool :
  public extends<GaudiTool, ITracksFromTrackR> {
  ranges::v3::any PrVeloUTTool::createState() const {
    PrVeloUTEventState eventState;
    ...
    return eventState;
  }
  StatusCode tracksFromTrack
    (..., ranges::v3::any& eventStateAsAny) const {
    PrVeloUTEventState &eventState =
      ranges::v3::any_cast<PrVeloUTEventState&>
      (eventStateAsAny);
    ...
    for(auto& ah : eventState.m_allHits) ah.clear();
```

After - Usage

```
m_veloUTTool = tool<ITracksFromTrack>(...);
auto eventState = m_veloUTTool->createState();
for(const auto& veloTr: inputTracks) {
  m_veloUTTool->tracksFromTrack(..., eventState);
  ...
}
```

This is the end

Questions ?

http://cern.ch/sponce/LHCbFutureHandsOn