Condition and Run Handling in Gaudi Mila

...

Benedikt Hegner
(CERN EP-SFT)

3.8.2016

Introduction into the Problem

- See Hadrien, who did a nice introduction for conditions data!
- https://twiki.cern.ch/twiki/bin/view/Gaudi/CommonConditionInfrastructure
- Proposes a pragmatic first step by re-ordering of events and not having multiple runs active in the event loop

What happens if we allow multiple events and conditions in active in the event loop:

1. Options for handling/accessing multiple conditions
2. Straw-man interface
3. Possible Implementation
4. Extensions to run and lumi data in general
5. How to handle run transitions and boundaries

This presentation is food for thought, not the final design or implementation

Pointers to Conditions

As often I start with the **user view**

In current vanilla Gaudi pointers to conditions are initialized once.
And the values behind these pointers are updated as necessary

Obviously that does not work any more with multiple runs active.

What are our options?

1. Never ever have multiple runs active in the event loop
2. Re-set the pointers under the hood in ::sysExecute
3. Use smart pointers that reset themselves
4. Let the user go to the condition service during each ::execute

Dispatching access to different events and runs

If we want to provide access to concurrent conditions, we have to forward an algorithm via smart pointers to the proper data

The hive data handles and whiteboard do that already via the **event slots**

One could do exactly the same of conditions, using **condition slots**

```
ConditionsHandle<T> myData{self, "jetcalib", "<help>"};
…
myData->someMethod(); // using condition slot internally
```

where multiple events having the exact same set of conditions share a **conditions slot**

How would that integrate with the rest of the framework?

Managing Condition Slots

The management of condition slots is handled by the *ConditionsSvc*

On receiving a new event, the *EventLoopMgr* approaches the *ConditionsSvc* with the timestamp of the event:

```
evtContext.conditionSlot = conditionsSvc->prepare(timestamp);
```

If there is a condition slot that matches the time stamp

⇒ increase usage counter; return that slot

If there is no matching slot

⇒ prepare all conditions in a new/re-used slot

Once an event has finished processing, let the *ConditionsSvc* know.

```
conditionsSvc->release(conditionSlot)
```

Keep internal machinery invisible

With **external** interfaces like

```
ConditionsSvc::get<T>(std::string id, T& handle)
ConditionsSvc::prepare(timestamp t)
ConditionsSvc::release(conditionSlot slot)
```

and the **handle** interface exposed to the user code

```
void ConditionHandle<T>::ConditionHandle(Alg&,string id, string desc)
T& ConditionHandle<T>::operator*();
```

and the promise that

data are valid/consistent during the call to ::execute
(and the hope that the dereferencing is non-blocking)

the internals of the *ConditionsSvc* are not constrained whatsoever by this proposal.

Not even whether you use multi-objects or multi-stores in the background

Possible implementation for the ConditionsSvc 1/2

The *ConditionsSvc* keeps a fixed registry of conditions objects, being filled at `::initialize` depending on announced requests.

Each conditions object consists of two entities:

1. ConditionsData<T>
2. ConditionsUpdater

ConditionsData

Holds the data for the various condition slots in a vector
(in fact it holds smart pointers to the concrete data in a vector)

ConditionsUpdater

Knows about the conditions data, its validity and how to update it
There are two main types of update strategy:

1. DB reading
2. Calculating derived conditions from other conditions data

Can be configured as plugins via the standard Tool mechanism

Possible implementation for the ConditionsSvc 1/2

In this schema the `ConditionsHandle<T>` is nothing else than a smart pointer to the `ConditionsData<T>` retrieving the data for the *active conditions slot*

On `::prepare(timestamp)`
- Look for a matching slot with proper global IOV
- If not found “loop” (*) through all updaters and ask for filling the new slot
  - If valid IOV found in internal data ⇒ copy pointer to data into new slot
  - If no valid IOV found in internal data ⇒ fetch or compute proper conditions

On `::release(slot)` the service
- goes through all conditions data and frees the slot
- smart pointers to the concrete data take care of the rest

(*) in fact the derived data form a DAG which has to be handled smartly

Dealing with Concurrent Runs and Lumis

Extending the concept for any run/lumi-dependent data

One of the dreams is to have “stateless” (well, re-entrant) algorithms. This is destroyed by *frame*-dependent (*) data like:

1. Statistics counters (#events seen, efficiencies, ...)
2. Monitoring histograms
3. ...

⇒ we should factor them out and **separate algorithms and data**
⇒ frame-dependent data are hidden behind special handles and live outside the algorithm

(*) *frame* is a more generic term for run, lumi, ...

Using frame slots

Generalize and replace the *conditionSlot* with a **frameSlot:**

- ➔ the number of frameSlots defines how many lumis/runs/conditions can be active at a time
- ➔ access to run/lumi-specific data is forwarded to the respective slot
- ➔ both event and frame slots are managed by the *EventLoopMgr*

Requires a few new handle types, including

1. *Frame*Specific<T> for frame specific data
2. *Frame*Read<T>, *Frame*Write<T> for data stored in run/lumi blocks

From user perspective:

```
// event specific
ReadDataHandle<ElectronCollection> m_electrons{...};
// conditions
ConditionsHandle<JetCalib> m_jetCalib{...};
// run data
RunSpecific<Counter> m_events;
```

Notification about begin/end of frames

The incidents as implemented now are not context-aware

⇒ need to add a context to such messages

Once a new frame enters the event loop, the EventLoopManager picks a frameSlot, and notifies clients via ::begin*Frame*(frameSlot)

Once all events of a frame left the event loop, the EventLoopManager notifies all clients via ::end*Frame*(slot) and is then free to re-use the slot.

It is not entirely trivial, as a certain order may be needed, e.g. run data has to written into the run record ***before*** *the data are written to disk.*

⇒ no new requirement w.r.t. current framework though

A full example

Reading new run and a few events from input file triggers the following behaviour:

The EventLoopMgr

1. Opens a new frame slot for that run
2. Triggers `beginRun(frameSlot)` for all relevant components
   a. ConditionsSvc prepares required data
   b. Algorithms initialize run-specific data
1. Opens eventSlots for all events and pushes them to the scheduler
2. Upon completion of all events in the run trigger `endRun(frameSlot)`
   a. Algorithms flush run-specific data
   b. *ConditionsSvc* does garbage-collection
3. Frees the frame slot

Comparison to CMS' solution

This proposal: separate algorithms and data

⇒ algorithms are frame-independent

⇒ data are multiplied for multiple frames

⇒ access to data is dispatched at runtime

⇒ #concurrent frames limited by #slots

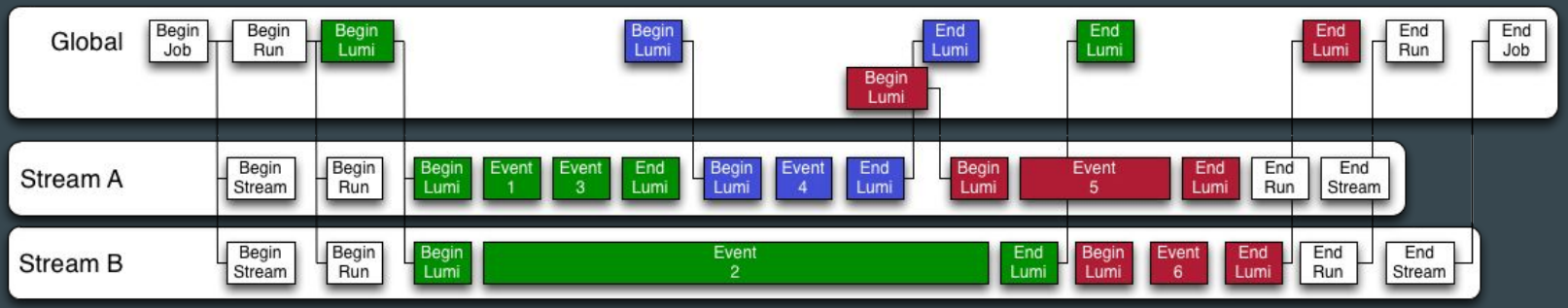
CMS: split application into "streams" that have a guaranteed run ordering

⇒ algorithms with run-specific states are tied to a given stream

(required deriving from different classes to pick proper behaviour)

⇒ algorithms and states therein are multiplied for multiple streams

⇒ #concurrent runs limited by #streams



Summary

Proposed a solution for concurrent conditions:

1. Described the problem
2. Provided a minimal interface for handles and *ConditionsSvc*
3. Described one possible implementation of this interface

Proposed a solution for handling runs and lumi-sections in general

1. Even better separation of algorithms and data
2. Context-aware pointers and messages

Will document all this ideas properly for the workshop

Including the semantics for algorithm clones and run/lumi data merges

Looking forward to a discussion/sprint during the workshop on

"Concurrent Runs and Conditions"