Assumptions Broken in a Multithreaded Framework: Solutions and Ideas

D. Piparo for the Concurrent Gaudi Team

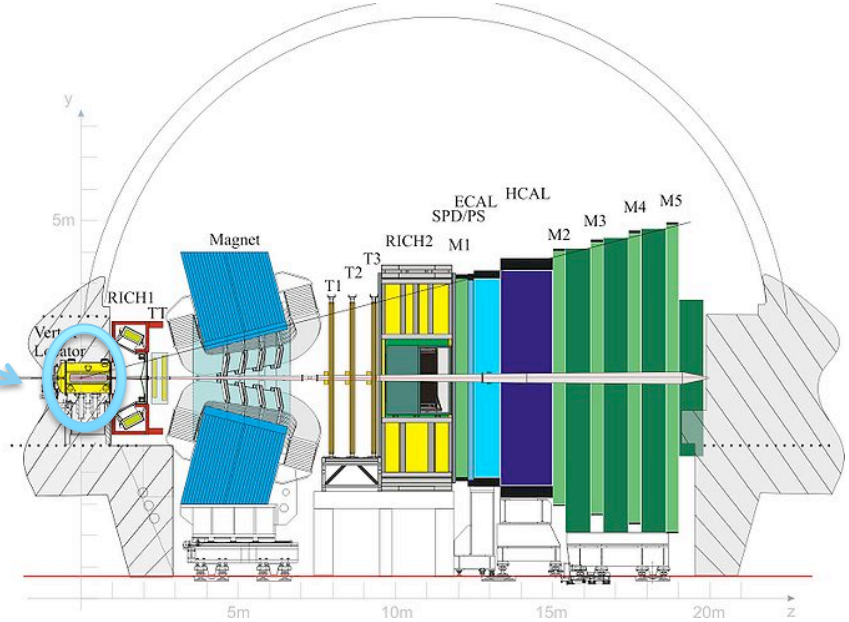
Concurrency Forum Meeting 23-4-2013

Our Goal

Provide a refurbished Gaudi framework which supports concurrent execution of algorithms and simultaneous processing of multiple events

- Pragmatic approach: start from slice of a real LHCb reconstruction workflow (called MiniBrunel in the following)
  - ~20 algorithms and associated tools: raw decoding and Velo tracking

MiniBrunel span within the detector



See backup for useful links about the project.

General Strategy

- Classify and document issues encountered during this effort
  - Build a "matrix of costs" - assess the size of the effort that would be required to migrate the complete LHCb stack
- Identify solutions and migration strategies
  - Not only thread safety issues: assumptions valid in the serial case are broken
  - Operate on existing large codebase
  - Minimal changes of interfaces
  - Provide new components pluggable in the present infrastructure
- Timescale: provide all pieces for MiniBrunel parallel execution by the end of June

MiniBrunel: Data Dependencies

Control flow dependencies not displayed

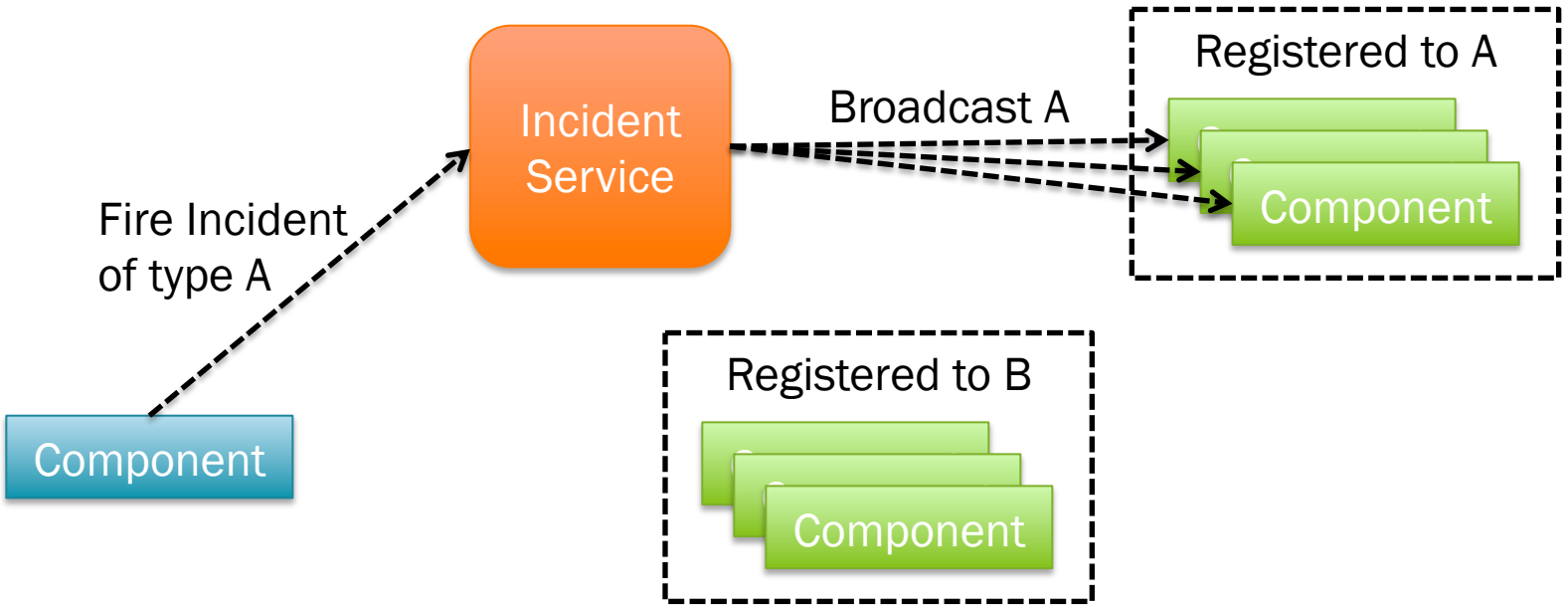
Solutions discussed today

1. **Incidents**
2. Serialisation of actions with a queue of closures
3. Thread unsafe resources: the I/O example
4. Caches

Incidents

Incident Service: central component of Gaudi managing *incidents* (signals)

- Components issue incidents with it
- Components are notified by it if registered a particular incident



Problem: what if an incident is linked to a state, e.g. a particular event?

Incidents: Dependency on a State

Example of a *stateful incident*:

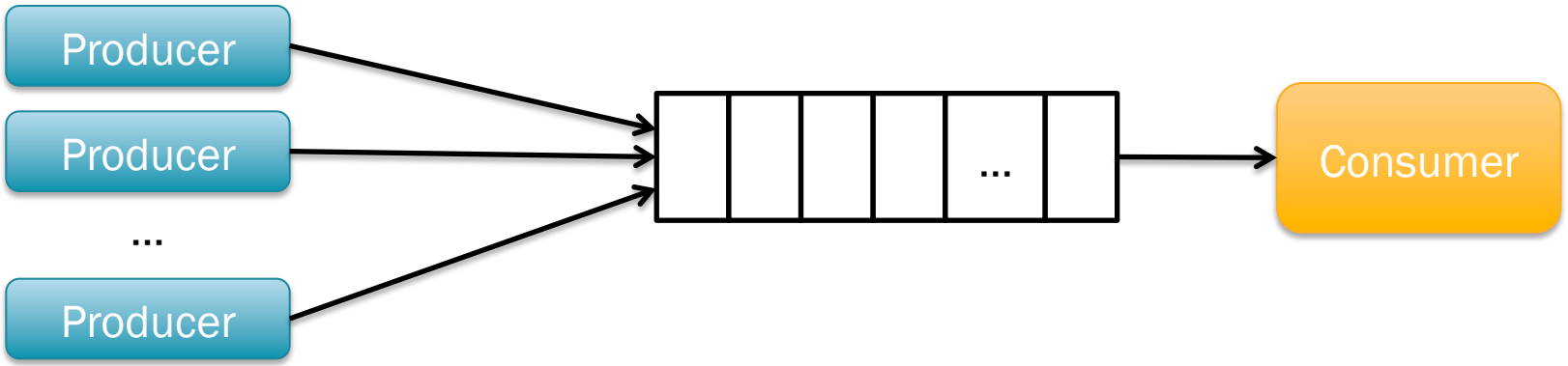
- "BeginEvent": issued by the event loop manager *before* algorithms' execution starts
- Concept not well defined anymore if multiple events processed simultaneously
- MiniBrunel: remove "reactions" to BeginEvent – move them to the event processing stage, steered by ordinary algorithms, collected in a "prolog sequence".
  - Implies understanding of the Physics behind the user code

Solutions discussed today

1. Incidents
2. **Serialisation of actions with a queue of closures**
3. Thread unsafe resources: the I/O example
4. Caches

Serialisation of actions: concurrent queue of closures

- Known pattern: lay out items in a queue in parallel to work them out serially
  - In our case: many producers possible, one consumer
  - "Message passing"-like approach
- Combine C++11 and TBB
  - C++11- function object + reference environment ("closure")
  - TBB – provides the `tbb::concurrent_bounded_queue<T>` class*
    - Concurrent push-pop of items
    - `void ::pop(T& dest)`: fill `dest` with last item, otherwise block (spins for a while and then puts thread to sleep)



* http://threadingbuildingblocks.org/docs/doxygen/a00268.html

Serialisation of actions: concurrent queue of closures

The idea

- Define an "action" as a `std::function<void()>`
- Different entities produce and push actions in the queue, a listener method (executed in a separate thread) consumes (calls) these actions.
- Handy way to take an action only when needed without wasting cpu cycles
  - No "`while(true){ … do checks and work …}`" construct
- Downside: cannot retrieve return value of the stored closures

Example 1: Inert Message Service

- Need to print messages coming from components running in different threads
  - Arrival of output to print intrinsically asynchronous!
- Previous TBB Message Service*: excellent and functional
  - But implied creation of a TBB thread pool very early (message logging: first component needed in the Gaudi application)
  - TBB does not allow to resize a pool once created!
- Nota bene: several implementations can live behind the same interface in Gaudi!

A new class

which allows to:

1. Wrap the call to the class' message printing methods into closures
2. Push the closure to an internal `concurrent_bounded_queue`
3. Execute a "listener" method of the same class in a separate thread to pick up the closures and run them

A drop-in replacement, transparent for the users.

* https://indico.cern.ch/getFile.py/access?contribId=4&resId=0&materialId=slides&confId=214319

Example 1: Inert Message Service

```
std::thread m_thread;
[…]
StatusCode InertMessageSvc::initialize() {
[…]
m_thread = std::thread (
        std::bind(&InertMessageSvc::m_activate, this));
return StatusCode::SUCCESS; }
```

Start listener in a separate thread

```
void InertMessageSvc::m_activate(){
m_isActive=true;
messageActionPtr thisMessageAction;
while (m_isActive or not m_messageActionsQueue.empty()){
    m_messageActionsQueue.pop(thisMessageAction);
    (*thisMessageAction)(); }
    }
```

Wait for actions to execute

Full code at http://lcgapp.cern.ch/git/GaudiMT/tree/GaudiCoreSvc/src/MessageSvc

Example 1: Inert Message Service

```
typedef std::function<void()> messageAction;
typedef std::shared_ptr<messageAction> messageActionPtr;
tbb:concurrent_bounded_queue<messageActionPtr>
                                                m_messageActionsQueue;
[…]
void InertMessageSvc::reportMessage(const Message& msg,
                                    int outputLevel) {
  m_messageActionsQueue.push(
  messageActionPtr(new messageAction([this,msg,outputLevel] (){
     this->i_reportMessage(msg, outputLevel)
     })));
}
[…]
```

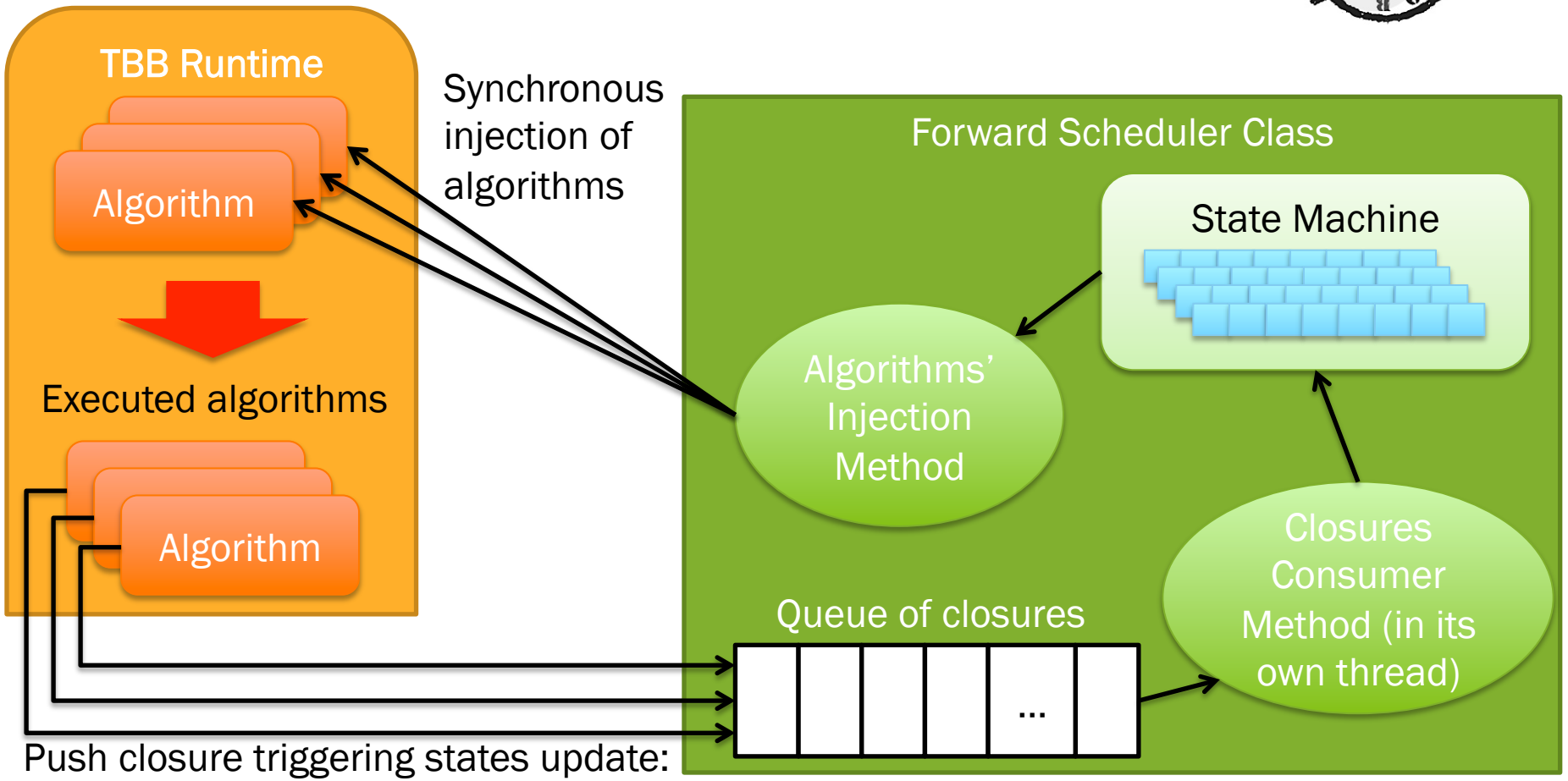
Create closure and push to queue

```
void InertMessageSvc::m_deactivate(){
  if (m_isActive){ // This would be the last action
    m_messageActionsQueue.push(messageActionPtr(
             new messageAction([this](){m_isActive=false;})));
    } }
```

Stop waiting for actions

Example 2: Forward Scheduler

- Component that submits to TBB runtime algorithms according to their data and control flow dependencies
- Absorb the asynchronous finishing of submitted tasks
- Update internal algorithms' state machine accordingly

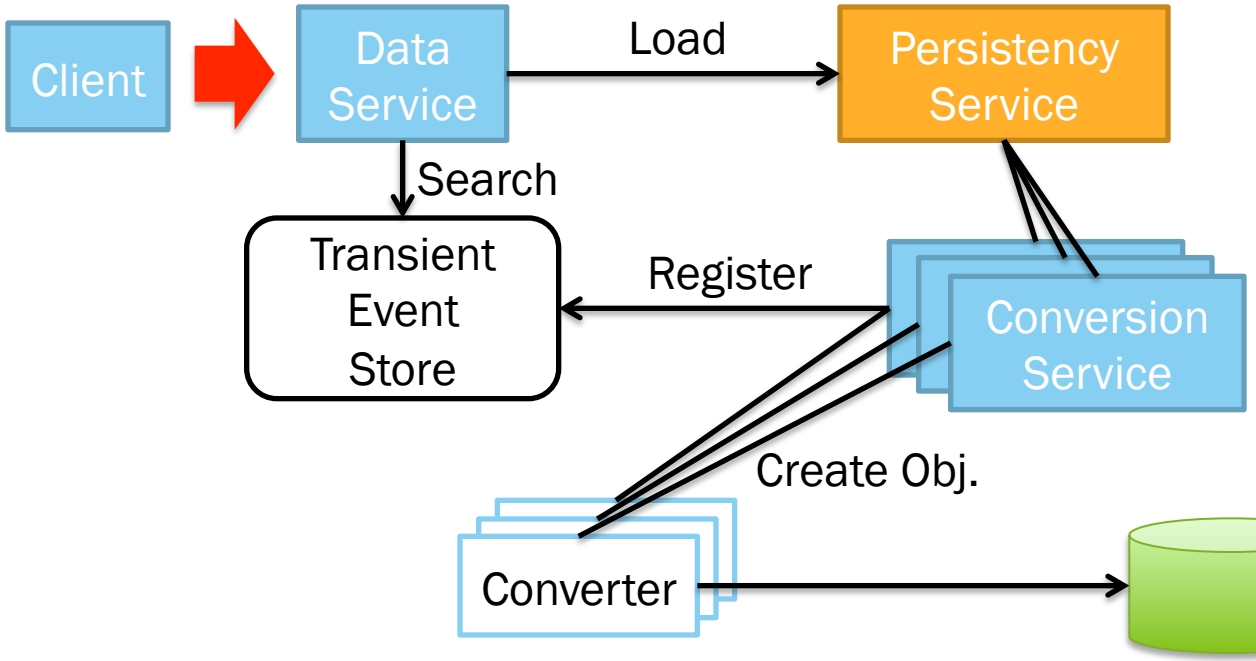


Solutions discussed today

1. Incidents
2. Serialisation of actions with a queue of closures
3. **Thread unsafe resources: the I/O example**
4. Caches

I/O thread unsafety: containment strategies

- Two kinds of persistency mechanisms encountered: ROOT and MDF
- MDF: Master Data Format for LHCb raw banks
- Commonality: no simultaneous read, write, read-write ops presently possible!



This breaks when several clients ask for data simultaneously

- Non-clonable algorithms populate and flush the transient event store
  - Clients: a single "ReadAlgorithm" and "OutputStreams"
  - Special control flow dependencies: "prologue" and "epilogue" algorithms' sequences
  - Only one algorithm of this category can be in flight at the time!

Solutions discussed today

1. Incidents
2. Serialisation of actions with a queue of closures
3. Thread unsafe resources: the I/O example
4. **Caches**

Caches and their classification

- Code base: full of caches
- Valid use cases:
  - One time initialisation (e.g. LHCb::RawEvent)
  - Result caching for multiple independent accesses (e.g. MC-truth matching)
- Invalid, discouraged, unofficial use case: communication channels between algs
  - Topic too complex to be treated here

Ways to address caches

- Check whether they are actually needed!
- One time initialisation: make the object a data product and use the event store
- Result caching: provide one separate cache per event processed simultaneously

Caches and their classification: the Idea

Logical building block for event parallelism: "processing slot" (# of events in flight)

The Idea

Create one object per slot and dispatch them using a smart pointer


```
template<class S> class IContextSpecific {}
[…]
S* operator->() const { return m_reg[thread_global_slot_id]; }
[…]
}
```

For Services and Tools the solution can be implemented centrally:

- Instance handling: respective plugin managers
- Existing SmartIF<T> pointer could be extended: minimise impact on usercode

Caveat: single caching instances themselves may still need to be made thread-safe!

Conclusions

- Progressing at full steam towards the parallelisation of MiniBrunel
  - Scheduling is solved, all other issues had to be addressed
- Issues related to real use cases were analysed
  - Work on the framework and users' code at the same time
- Solutions found and applied to the cases studied
- Potentially re-usable patterns abstracted

Backup

Useful Material

Project Page on the Concurrency Forum Site:

http://concurrency.web.cern.ch/GaudiHive

Main Twikipage:

https://twiki.cern.ch/twiki/bin/view/C4Hep

Git Repository Web Interface:

http://lcgapp.cern.ch/git/GaudiMT/

Jira:

https://sft.its.cern.ch/jira/browse/CFHEP

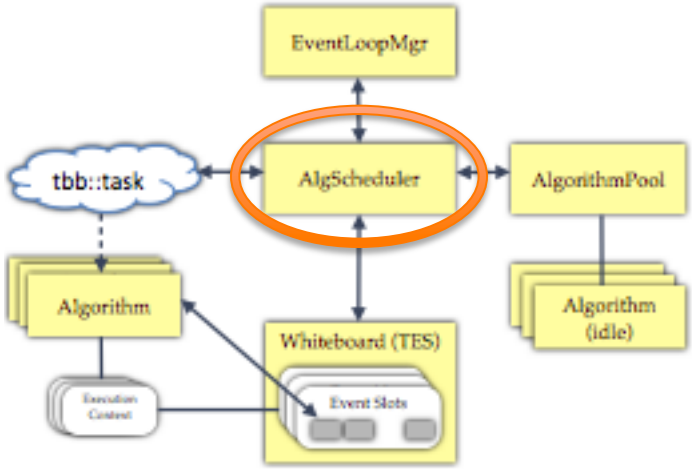
Weekly (Thursday 10:30 a.m., with phoneconf) Working Meeting Minutes:

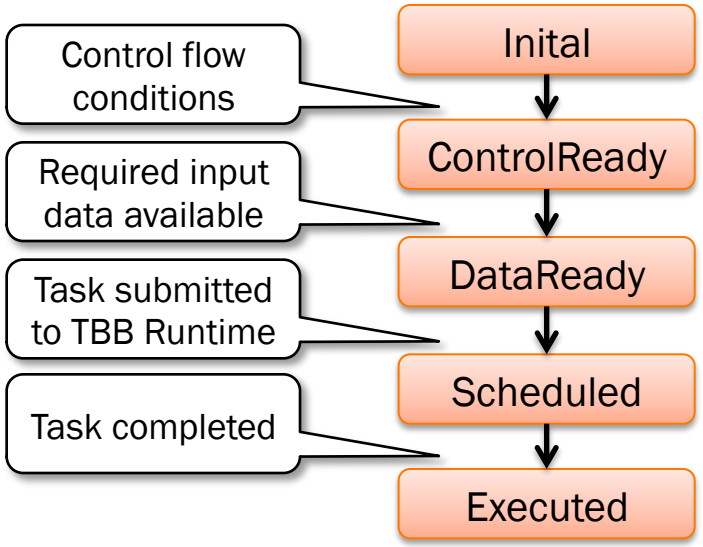
http://sync.in/k5XvRql9y9

The Forward Scheduler

Keeps the state of each algorithm for each event

- Simple finite state machine
- Receives new events from loop manager
- Interrogates whiteboard for new DataObjects
- Pulls algorithms from AlgorithmPool if they are available
- Encapsulate them in a tbb::task for execution
- Absorbs asynchronous events (e.g. arrival of finished tasks) with a thread safe queue of lambda closures (*actions*). Same pattern used for new message svc.





Algorithm Pool

Contains algorithms and coordinate them

- Gives away instances to run, retrieves ran algorithms
- Clones algorithms (via AlgManager)
  - Number depends on code re-entrancy: non re-entrant (1 copy only), non re-entrant (use n copies), fully re-entrant (re-use same instance n times)
- "Flattens" sequencers
- Allow for exclusive resource checking:
  e.g. if 2 algos using a non re-entrant external library, only one at the time can run.