Re-Entrant Algorithms in AthenaMT

Charles Leggett
June 6 2016

Glasgow TIM

Motivation

- AthenaMT allows us to **clone** Algorithms
  - multiple instances of the same Algorithm
  - the Scheduler can concurrently execute the same Algorithm in different events (in different threads) by using different clones
    - don't need to worry about (most) thread safety issues, since each thread gets its own copy, and they don't interfere with each other
- Cloning Algorithms allows us to balance memory usage with scheduling concurrency
  - more clones = more opportunities to run simultaneously
  - but, more clones = more memory
  - we can control the number of clones of any Algorithm at run time
- Re-entrant Algorithms allow us to run the same Algorithm concurrently in different threads, but minimize memory usage by only creating ONE Algorithm instance
  - win-win scenario!

Re-Entrant Algorithm Issues

- nothing good is free....

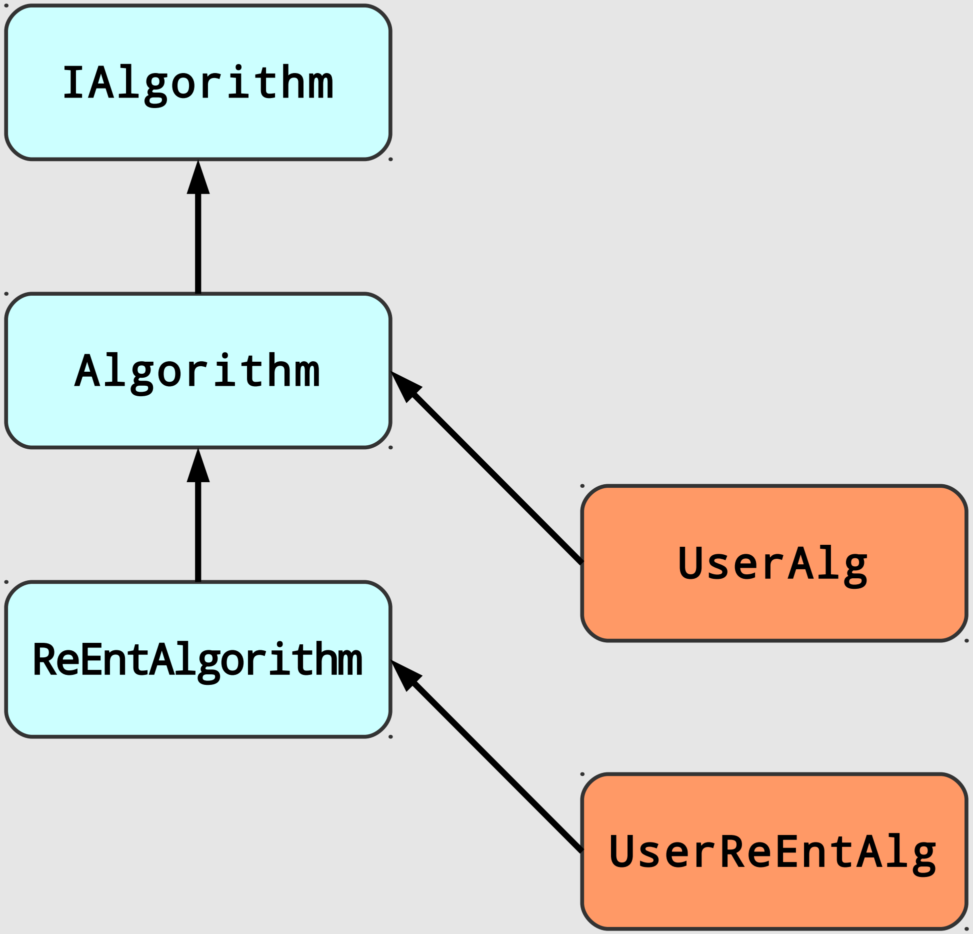
- Downside: Re-entrant Algorithms **MUST** be fully thread safe
  - normally cloned algorithms don't need to be completely thread safe as each thread gets it's own instance.
    - though they do need to avoid/protect thread hostile semantics like statics

- thread safety is **HARD** to implement, and re-entrant Algorithms will have to be (re)designed from the ground up

- `Algorithm::execute()` is **const** for re-entrant Algs
  - we'll give it a new signature `execute_R() const` to explicitly differentiate

- we also need to explicitly pass the EventContext
  - normally it's part of the Algorithm
  - `execute_R(const EventContext&) const`

Gaudi Class Hierarchy



Base Class Changes

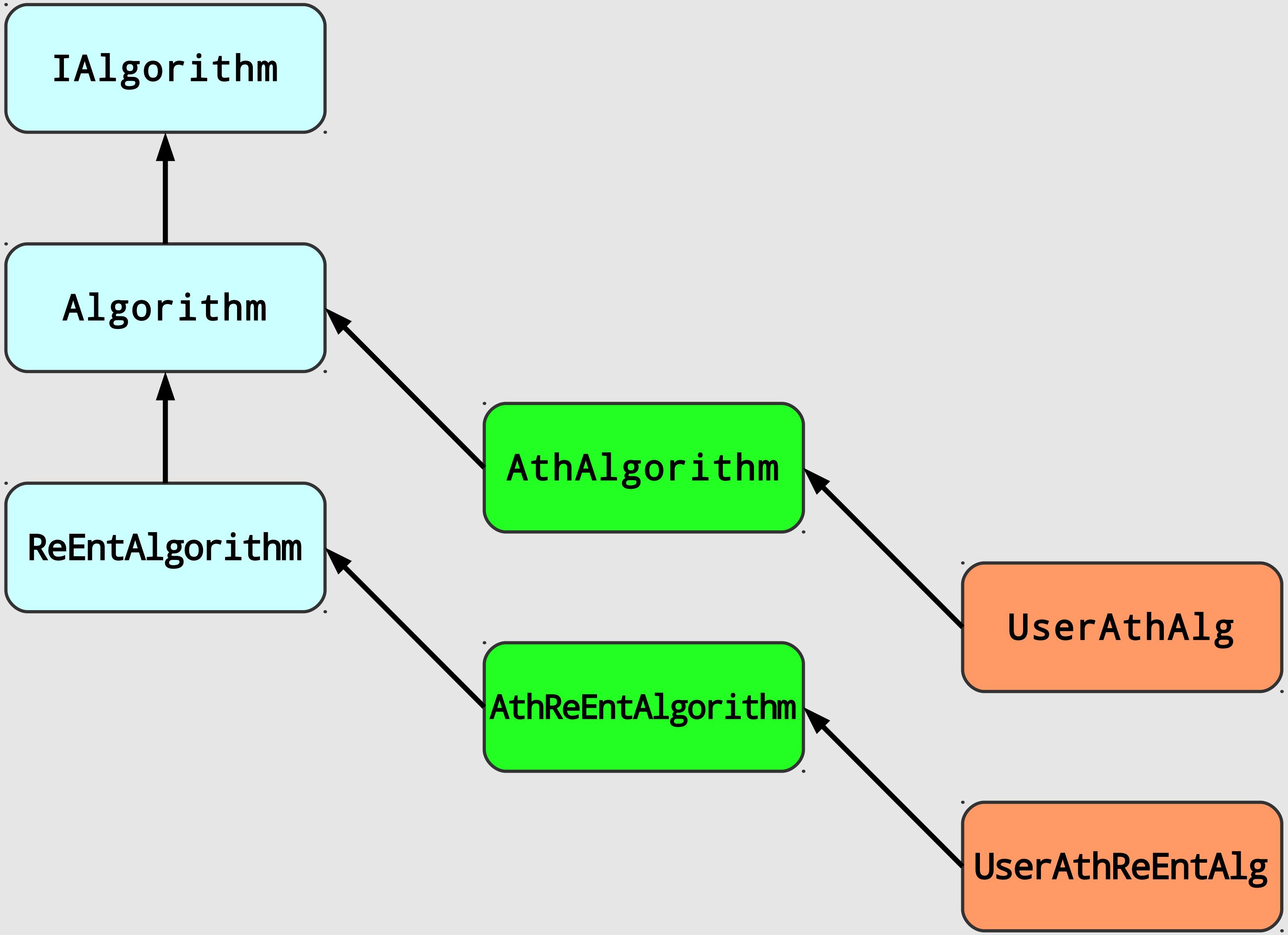
```
IAlgorithm
  bool isReEntrant() const;

  StatusCode execute_R(const EventContext&) const
  StatusCode sysExecute_R(const EventContext&)
```

```
Algorithm: public IAlgorithm
  bool isReEntrant() const { return false; }

  StatusCode execute_R(const EventContext&) const {
    return StatusCode::FAILURE;
  }
  StatusCode sysExecute_R(const EventContext&) {
    return StatusCode::FAILIRE;
  }
```

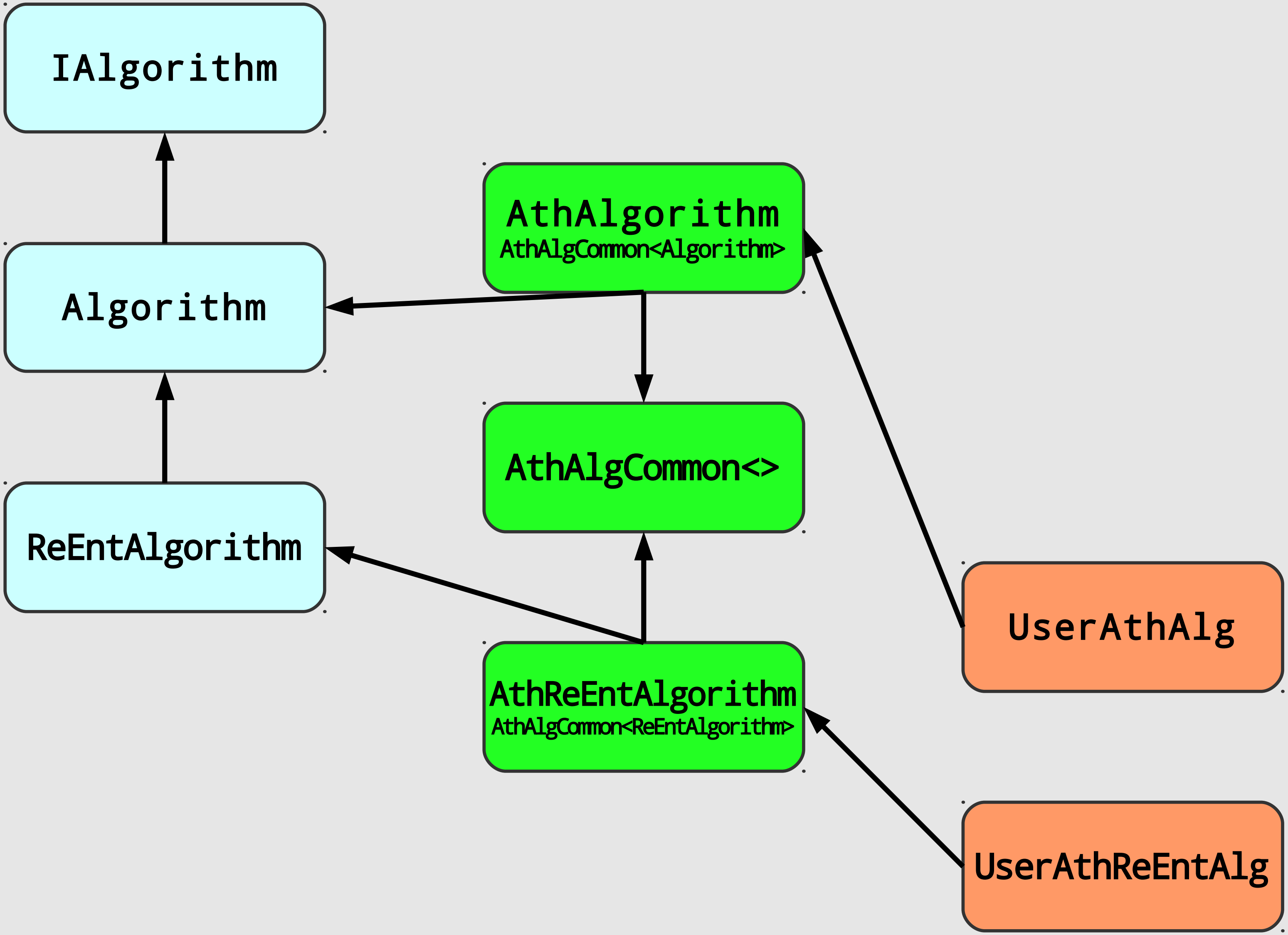
Base Class Changes

```
ReEntAlgorithm : public Algorithm
  bool isReEntrant() const { return true; }

  StatusCode execute() {
     return execute_R( Gaudi::Hive::currentContext() );
  }

  StatusCode sysExecute_R(const EventContext& ctx) {
      ...
      status = execute_R(ctx);
      ...
  }
```

AthAlg Class Hierarchy



AthAlg Class Hierarchy



Problems

- Algorithm base class keeps some event dependent status information as member data
  - filter passed flag
  - executed flag
- Other event dependent status info is kept in the EventContext (which is `const` when passed to the Alg)
  - event failed
- All this needs to be moved elsewhere

- New service **`AlgExecMgr`** which keeps track of:
  - execution state of each Alg in each slot
    - filterPassed, isExecuted, execStatus
    - `vector< map<AlgKey, AlgExecState> >` (one vector entry per slot)
  - overall execution status of the event
    - Success / AlgFail / AlgStall / Other err
    - `vector< EventStatus >` (one vector entry per slot)

How It Works

- AlgResourcePool
  - will only create one instance of a ReEntAlgorithm
  - when asked for an re-entrant alg instance by the Scheduler, will always return the same one

- AlgoExecutionTask
  - if alg is re-entrant, will call `alg->sysExecute_R(evtCtx)` instead of `alg->sysExecute()`
  - after execution, sets alg / event status via the AlgExecMgr

- Scheduler
  - sets/resets all alg / event status via the AlgExecMgr

User Re-entrant Algorithm

```
class MyReEntAlg : public AthReEntAlgorithm {
public:
  StatusCode initialize();
  StatusCode execute_R(const EventContext&) const;
  StatusCode finalize();

private:
  SG::ReadHandleKey<EventInfo> m_evt;
  SG::WriteHandleKey<HiveDataObj> m_wrh1;
};

StatusCode MyReEntAlg::execute_R(const EventContext& ctx) const {

  ATH_MSG_INFO("execute_R: " << index() << " on " << ctx);

  SG::ReadHandle<EventInfo> evt(m_evt);
  ATH_MSG_INFO("   EventInfo:  r: " << evt->event_ID()->run_number()
               << " e: " << evt->event_ID()->event_number() );


  SG::WriteHandle<HiveDataObj> wh1(m_wrh1);
  ATH_CHECK( wh1.record( CxxUtils::make_unique<HiveDataObj>
                         ( HiveDataObj(10000 )))
           );
  ATH_MSG_INFO("  write: " << wh1.key() << " = " << wh1->val() );

  return StatusCode::SUCCESS;


}
```

instance number (0..n) of the Alg

- always 0 for re-ent Algs

have to use VarHandleKeys, and create VarHandle on stack

How To Use It

- This is not yet in the regular AthenaMT build
- It usually gets built for one nightly a week
  - I send out notices to interested parties
  - you can tell which build it is by looking at the README file in the root directory of the Gaudi build area
- You can make your code work in all builds by protecting the appropriate bits with the `#ifdef REENTRANT_GAUDI` macro
- There's an example in AthExHive/HiveAlgR

Comments

- There's a Gaudi merge request (WIP) where the design/implementation can be discussed:
  - https://gitlab.cern.ch/gaudi/Gaudi/merge_requests/177