Managing Asynchronous Data in ATLAS's Concurrent Framework

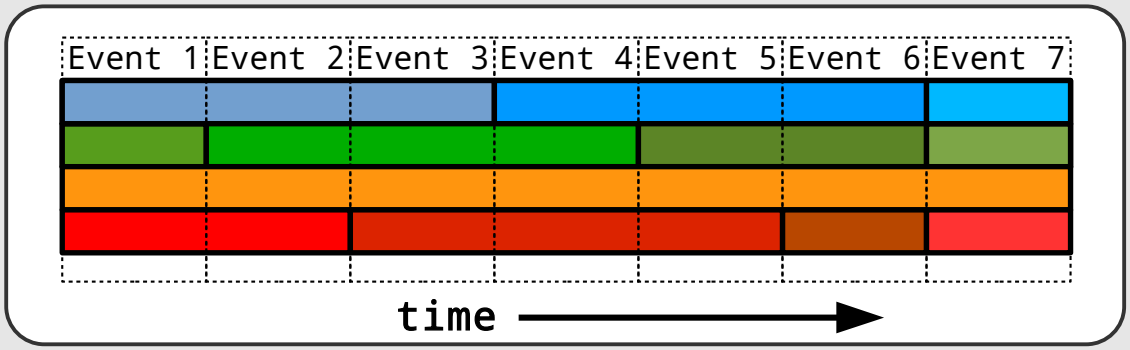
John Baines, Tomasz Bold, Paolo Calafiura, Jack Cranshaw, Andrea Dotti, Steven Farrell, Charles Leggett, David Malon, Graeme Stewart, Scott Snyder, Peter Van Gemmeren, Vakhtang Tsulaia, Benjamin Wynne

for the ATLAS Collaboration

ICHEP 2016

Asynchronous Data and Events

- Data that can change during the course of a job, but less frequently than once per Event (beam collision)
  - period for which any piece of data is valid is referred to as an **Interval Of Validity** (IOV)



- Classify into 3 broad types:
  - **Conditions**
    - eg high voltages, calibrations, etc
  - **Detector Geometry and Alignments**
    - eg: position changes
  - **Asynchronous Callbacks (Incidents)**
    - functions that need to be executed at non-predetermined intervals
    - eg: respond to a file open/close
- These are often inter-related
  - a condition change can trigger a callback

Asynchronous Data and Concurrency

- **Serial processing:**
  - one Event at a time
  - all framework elements process data from the same IOV
  - clients are blind to the IOV, and cache data locally
    - Services handle updating the data given the current Event time/ID. Only one copy (the current one) of any object needs to maintained
- **Concurrent processing:**
  - multiple Events processed simultaneously
  - different elements of the framework may have to process data from different IOVs
    - Services must now handle multiple versions of the same data, and deliver the appropriate one to each client, depending on which Event the client is processing
- AthenaMT allows cloning of Algorithms to enhance sub-event parallelism. Association between Algorithm instance and an Event is never guaranteed.

Further Complications for Concurrency

- Two types of Asynchronous Data
  - **Raw** – read directly from a database
  - **Calibrated** – after reading from database, the data is processed in some way by a function

- In Athena, this processing is managed by a Service (IOVSvc), and performed by special functional objects (usually AlgTools)
  - at the start of every Event, before Algorithms are processed, IOVs are checked and any necessary updates are triggered by the execution of these IOVSvc callback functions
  - shared instance of each AlgTool
  - the AlgTools tend to cache data

- The current callback AlgTools are **NOT** thread-safe, and even if they were made thread-safe, could **NOT** run with multiple concurrent Events from different IOVs due to the local caches
  - IOV infrastructure needs to be modified for MT

Requirements for AthenaMT

- **AthenaMT**: ATLAS's next generation, multi-threaded reconstruction/simulation framework
  - multiple simultaneous Events
  - sub-Event concurrency
  - multi-threaded
    - each Algorithm processes its Event in its own thread
- Try to minimize changes to User code
  - there's lots and lots of it!
  - avoid forcing Users to implement fully thread-safe code by handling most thread-safety issues at the framework / Services level
- Leverage MT design to minimize memory footprint
  - ATLAS reconstruction is very large
  - ratio of physical memory / CPU is constantly decreasing
- All access to Event data *via* DataHandles, which also declare data dependency relationship to the framework

Possible Solutions: Scheduling Barrier

- **Event Scheduling Barrier**
  - The framework only concurrently processes Events from within one IOV at a time. When a boundary is reached, it finishes processing all Events from the first IOV before starting to schedule Events from the next IOV

- **BENEFIT:**
  - completely transparent to Users
  - no code changes for Services
  - could make callbacks (inefficiently) thread safe with a big mutex

- **PROBLEMS:**
  - loss of concurrency / throughput if boundaries are frequent – processor is often idle
  - requires Events to be processed in order, or the ability to cache and shuffle incoming Events to avoid bouncing back and forth

Possible Solution: Multiple Stores

- **Multiple Conditions Data Stores**
  - Data is stored in EventStore-like structures, with one Store per concurrent Event
  - Clients access data via smart DataHandles, which point to the correct Store
  - Services update the data in the appropriate Store, depending on the associated Event

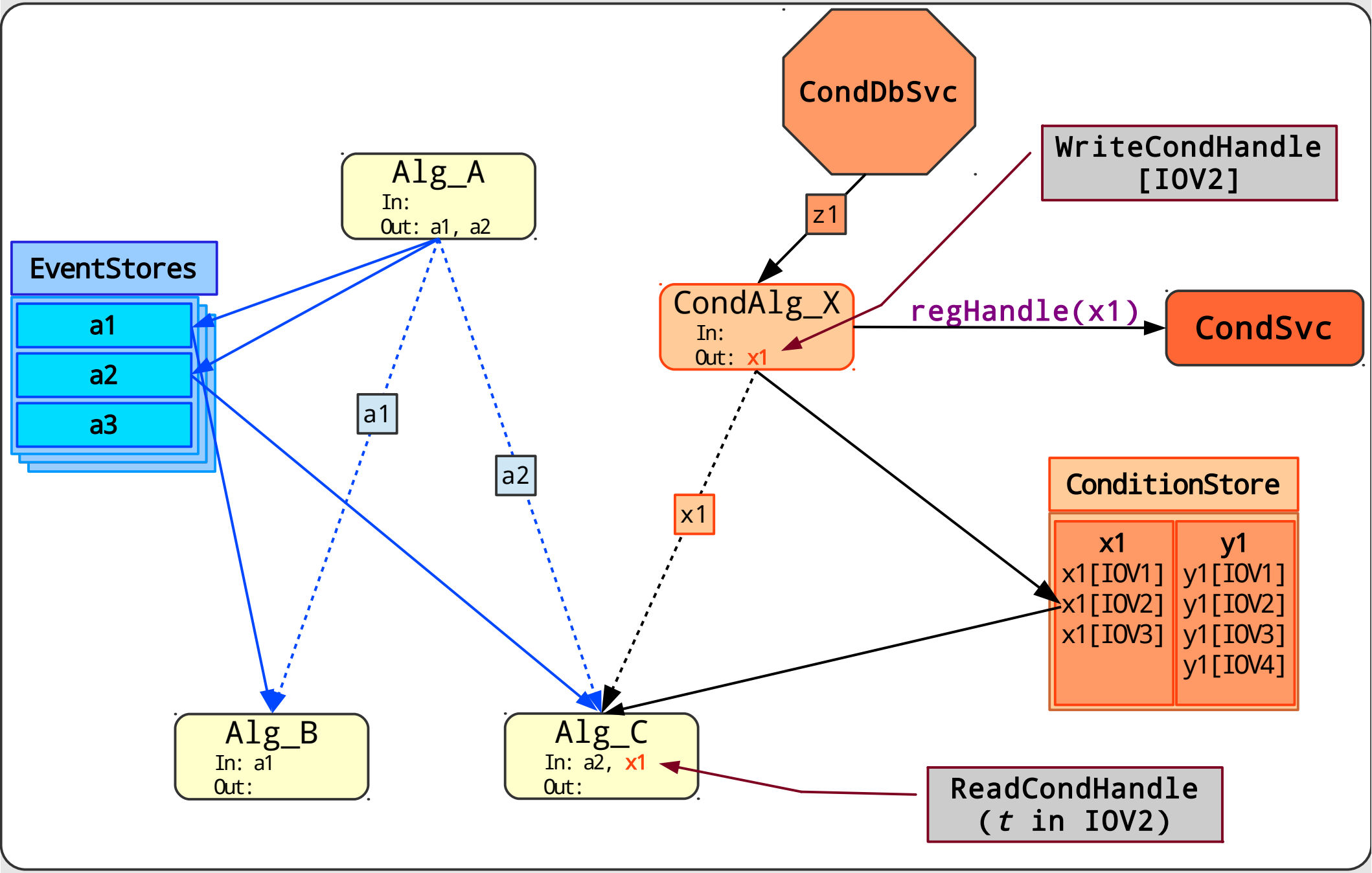
- **BENEFIT:**
  - only small changes needed to Client code (use of DataHandles), mostly hidden behind a layer of indirection

- **PROBLEMS:**
  - large memory overhead due to duplicate stores
  - duplication of re-calculations

Solution: Multi-Cache Condition Store

- **Single multi-cache Store for Conditions data**
- Each Store element is a **container** that holds multiple instances of the Conditons data objects (**ConditionContainer**), one per IOV
- Clients access the data via smart **ConditionHandles**, that point to the appropriate entry in the ConditionContainer objects for a given Event
  - ConditionHandles are constructed with an **EventContext** object
  - from the Client's point of view, these objects look like any other object in the EventStore (keyed with a unique identifier)
    - Client Algorithms declare a data dependency on the conditions data object
- Updating functions are scheduled by the framework, that load new elements from the DB, and perform any necessary computations
  - IOVSvc callback functions are migrated into **ConditionAlgorithms**
  - these Algorithms are only scheduled when they enter a new IOV

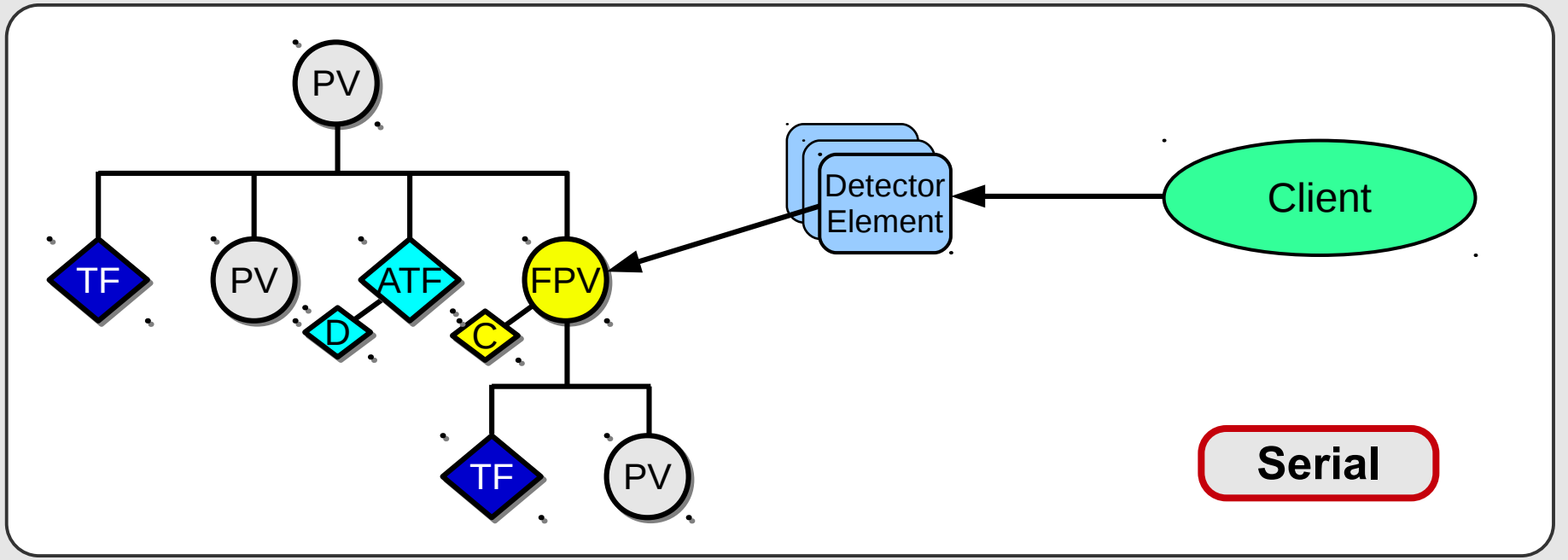
ConditionHandles

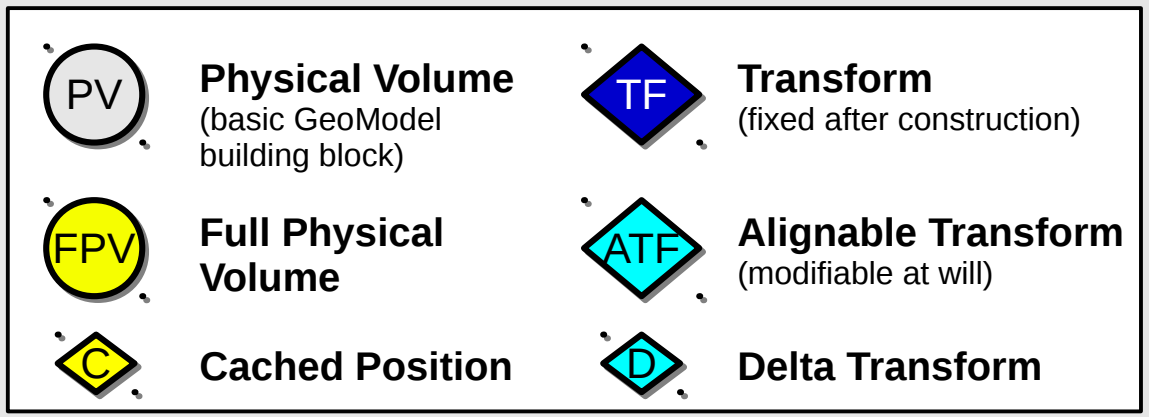


Memory Management

- While this makes optimal use of memory (no duplication of objects), the store will continue to grow with time
- Depending on memory constraints, may become necessary to perform garbage collection
  - prune ConditionContainers of old, unused entries
  - only keep N copies
  - keep reference count of which entries are in use, purge old entries

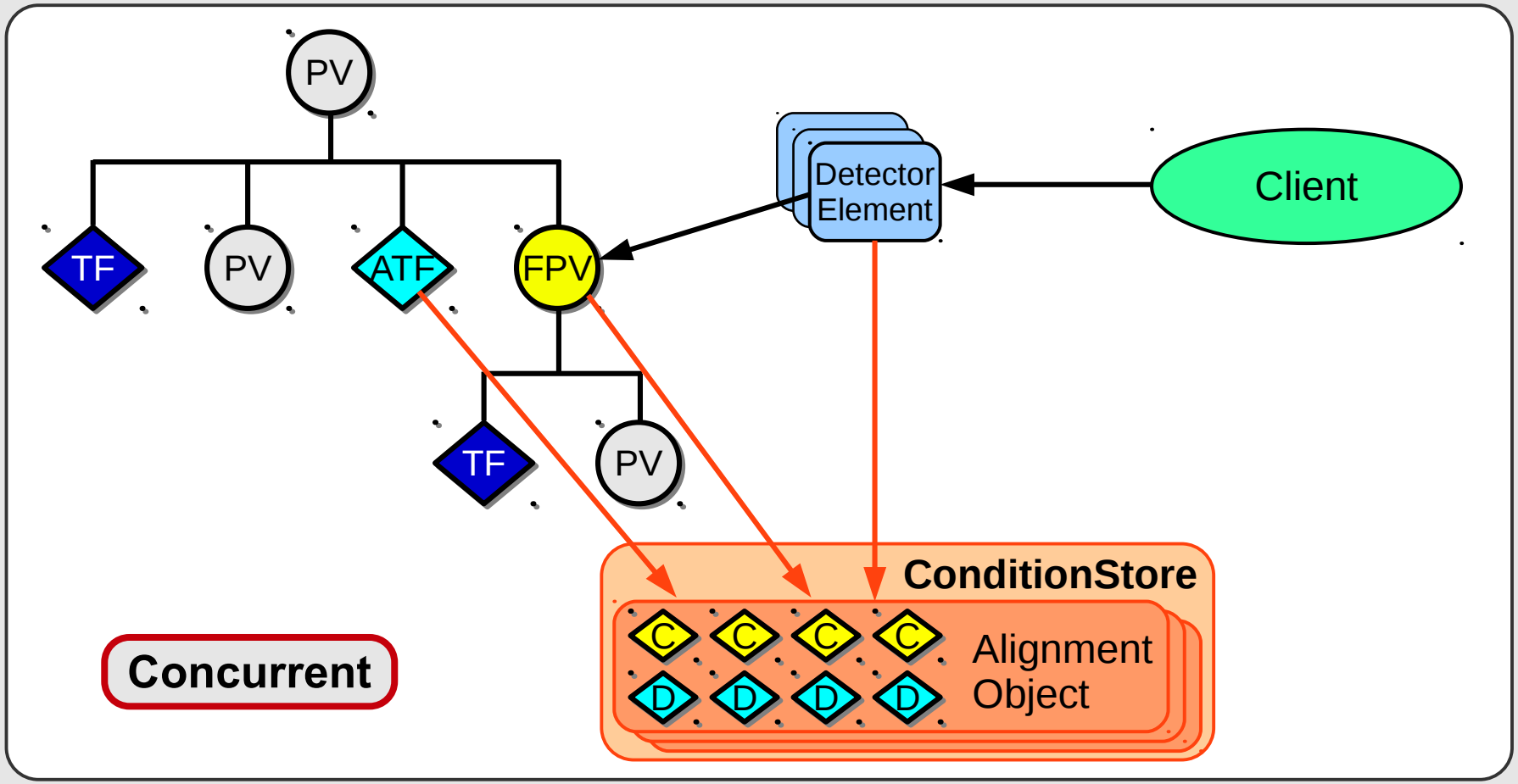
Detector Geometry Alignment





- **GeoModel** tree is not exposed to Detector Description clients
- Readout geometry layer consists of subsystem specific **Detector Elements**
- Each Detector Element has a pointer to **Full Physical Volume**

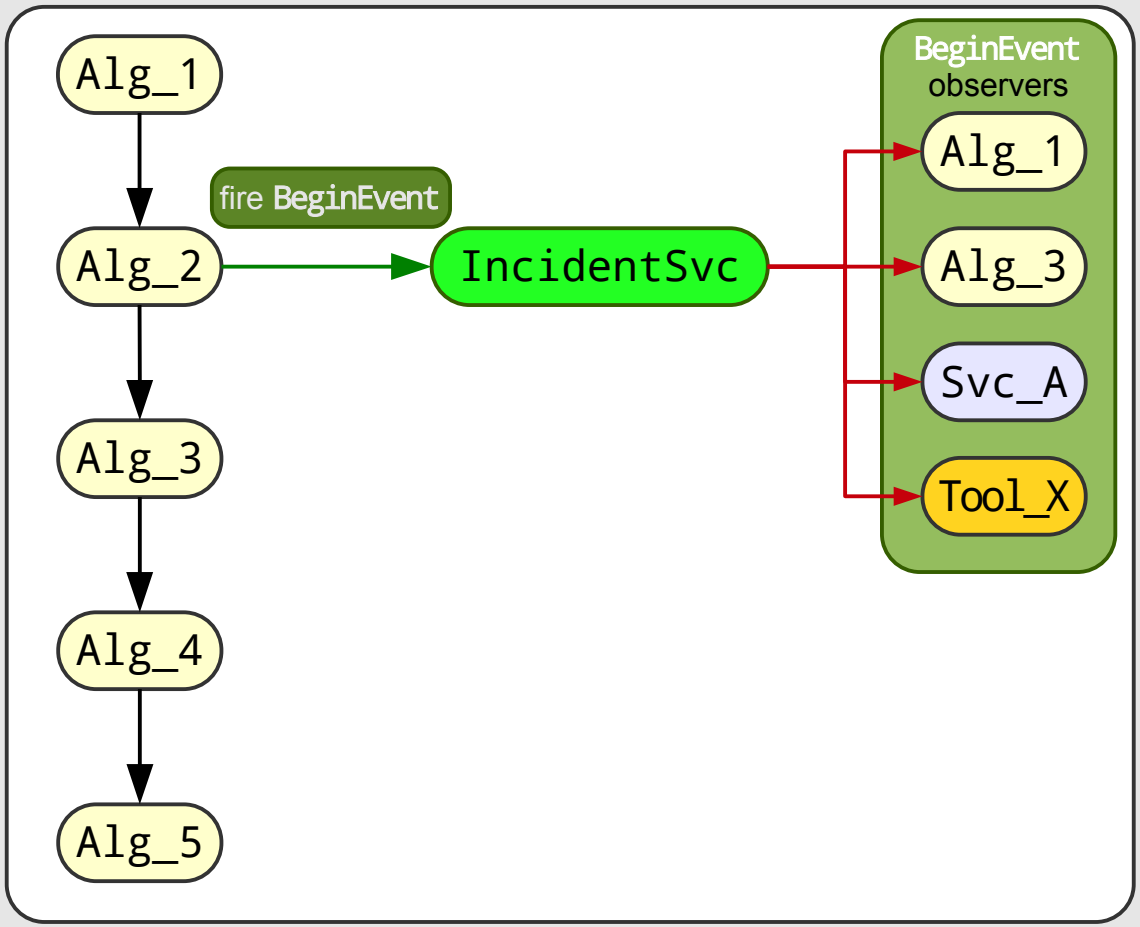
Geometry Alignments in AthenaMT



- The **Alignment Object** is a regular **ConditionContainer**, so it should be handled as any other Conditions Object in AthenaMT
  - Created by a **ConditionAlgorithm** (replacement of current callback function)
  - Accessed from the FPV and ATF via **Conditions Handle**
- By making Detector Elements aware of the Alignment Objects we can make the transition transparent to Detector Description clients

Asynchronous Incidents

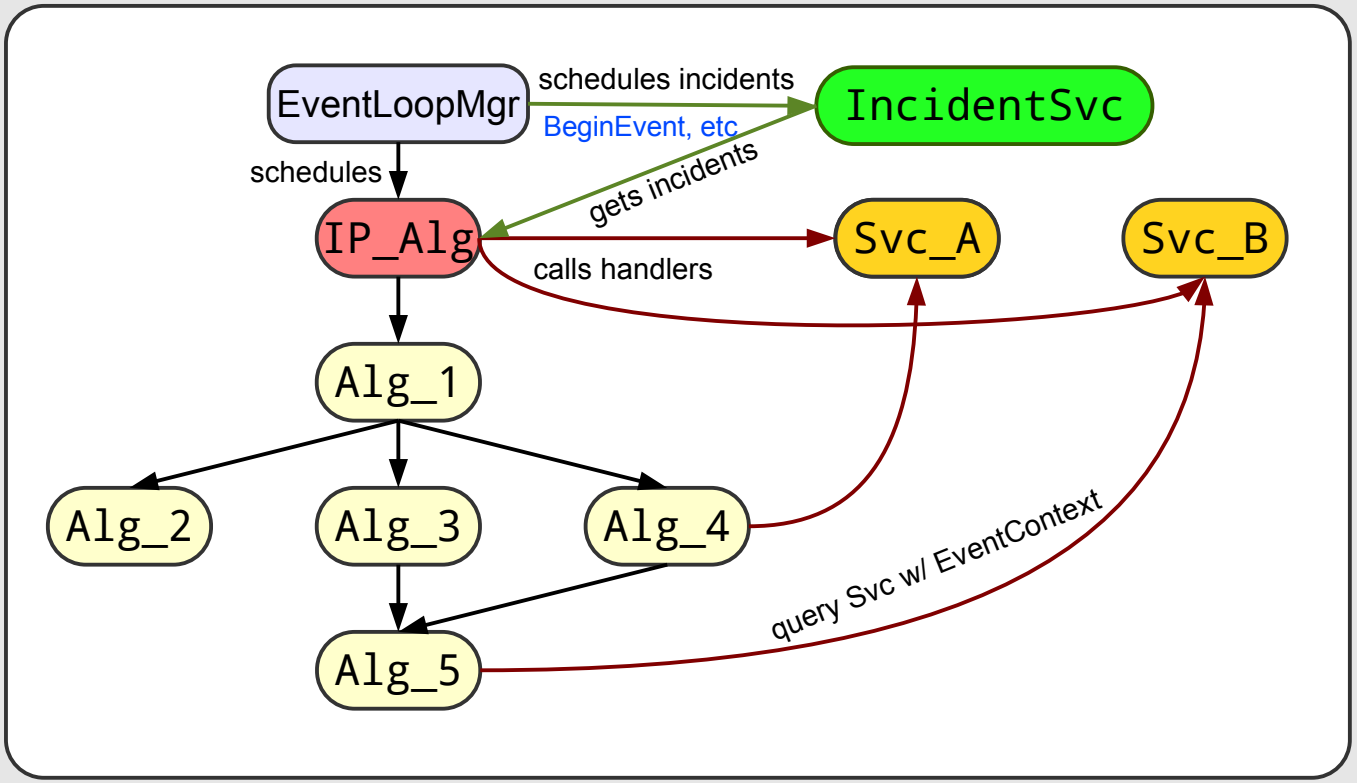
- **IncidentSvc**: manages asynchronous callbacks for clients which register as observers to specific events
  - *eg*: `BeginEvent`, `EndInputFile`, `MetaDataStop`
  - very flexible: callbacks can be triggered at any time
  - Clients can be anything: Algorithms, Services, Tools



Absolutely disastrous in an environment with multiple concurrent events, and multiple instances of each Algorithm

IncidentSvc in AthenaMT

- Study: IncidentSvc is overused / abused
  - mostly fired **outside** of the event loop
  - Incidents can be re-classified as discrete state changes
- Incidents become ***schedulable***, managed by framework
  - Incident handlers / observers become discrete Algorithms, that interact with Services which are aware of the EventContext



Conclusions

- Managing Asynchronous data in a concurrent environment will require a paradigm shift
  - no solution is fully transparent or plug-and-play, unless we choose to sacrifice concurrency and performance
  - dealing with multiple threads as well as multiple concurrent events is doubly challenging
- Have been able to minimize impact on User code via strategic modifications at the framework and Service level
- New versions of all three aspects of Asynchronous Data and Event infrastructure have been implemented, and migration of client code is ongoing, in conjunction with universal migration to DataHandles
  - so far, migration has been relatively straight-forward, and anticipate finishing by end of 2016