Common condition infrastructure

Hadrien Grasland, Benedikt Hegner

This project has received funding from the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 654168.

Motivation

Previously in GaudiHive...

- Gaudi was designed for single-core CPUs
- One Gaudi process per CPU core won't scale[1]
  - Excessive RAM usage caused by data duplication
- One event at a time won't scale[2]
  - Limited intra-event concurrency
- Concurrent events mean concurrent conditions
  - Legacy condition code not designed for this

[1] G. Stewart, “Overview and Status for ATLAS Phase I Software Upgrades”, 2015-12-01
[2] I. Shapoval, PhD Thesis, 2016-03-18

Two opinions

- Experiment developer will conclude:
  - Our custom condition code needs fixing
- Framework developer will conclude:
  - Conditions not an experiment specific concern
  - Gaudi should be taking care of this for you
- These viewpoints do not seem incompatible

A vision for Gaudi

- Gaudi should be aware of conditions
- It should provide a standard interface to them
- It should not dictate every detail
  - Can provide architecture + default implementation
  - Should enable a progressive migration from current experiment-specific infrastructure
  - And must allow experiment-specific tuning

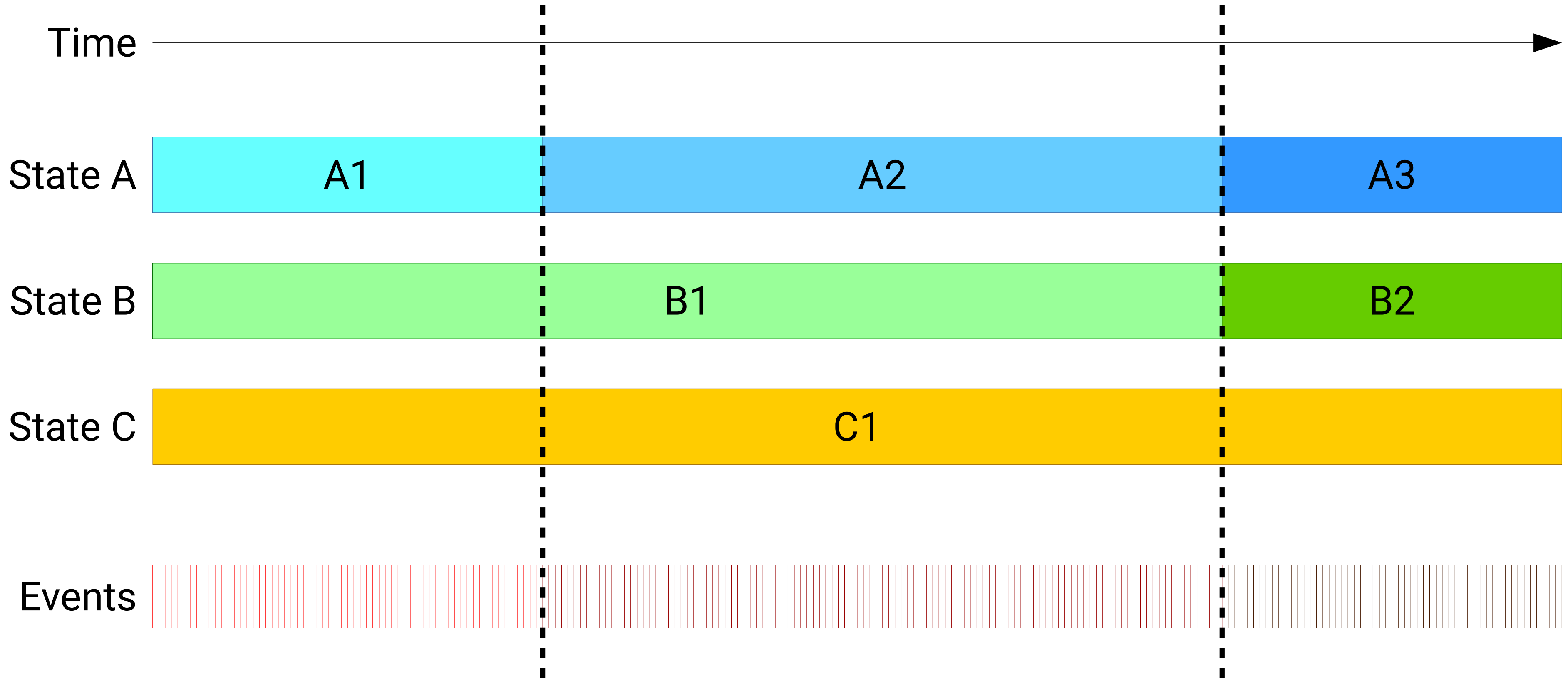
Requirements & design

Problem statement[3]

- **Detector state** is used during event processing
- It can be decomposed into sub-components
- Some of these are **time-dependent**
  - Time evolution modeled as sudden changes
  - Versioned data with Intervals of Validity (**IoV**)
  - We call a version of a state component a **condition**
- Must load/compute conditions as needed

[3] Iterated from https://twiki.cern.ch/twiki/bin/view/Gaudi/CommonConditionInfrastructure

A visual representation



(Caveat: Events are interleaved during parallel processing)

Problem statement

- **Detector state** is used during event processing
- It can be decomposed into sub-components
- Some of these are **time-dependent**
  - Time evolution modeled as sudden changes
  - Versioned data with Intervals of Validity (**IoV**)
  - We call a version of a state component a **condition**
- Must load/compute conditions as needed

Condition access

- Consensus: Use smart pointers/data handles
- Must account for data versioning
  - Should support concurrent condition storage
  - Could use a piece of EventContext to tell which conditions should be used
- Must be able to choose/swap storage backend
  - Many possibilities: DetectorStore(s), ATLAS' ConditionStore, DD4Hep...
- Must integrate with existing infrastructure

Storage requirements

- Cannot hide every implementation detail
  - Concurrent storage has an impact on clients
    - Large code changes required (all singletons must go)
    - Only worthwhile if condition switches frequent/expensive
    - As a specialized optimization, should be **optional**
  - Bounded RAM usage → bounded storage capacity
  - Must think about **garbage collection** from day one
  - Thread **synchronization** can become a bottleneck

Problem statement

- **Detector state** is used during event processing
- It can be decomposed into sub-components
- Some of these are **time-dependent**
  - Time evolution modeled as sudden changes
  - Versioned data with Intervals of Validity (**IoV**)
  - We call a version of a state component a **condition**
- Must load/compute conditions as needed

How do you measure time?

- Clocks
  - Intrinsic precision?
  - Synchronization?

- Atomic counters
  - What are you counting?
    (Events? Runs? Lumiblocks?)
  - How are you counting it?
    (Un-/signed? 32-/64-bit?)

→ Actually an experiment-specific mixture

A timing abstraction

- Experiments define **time points**
  - May have multiple internal representations
  - Different representations are not comparable
  - Give partial order & prioritize representations
- All time points follow a **common interface**
  - Check if time points are comparable
  - Compare time points with one another
  - Separate representations when needed

Modeling IoVs

- From time points, one can build time intervals
  - Defined using two comparable time points
  - Can tell if a time point falls into an IoV
  - Can be intersected to compute global IoV

Problem statement

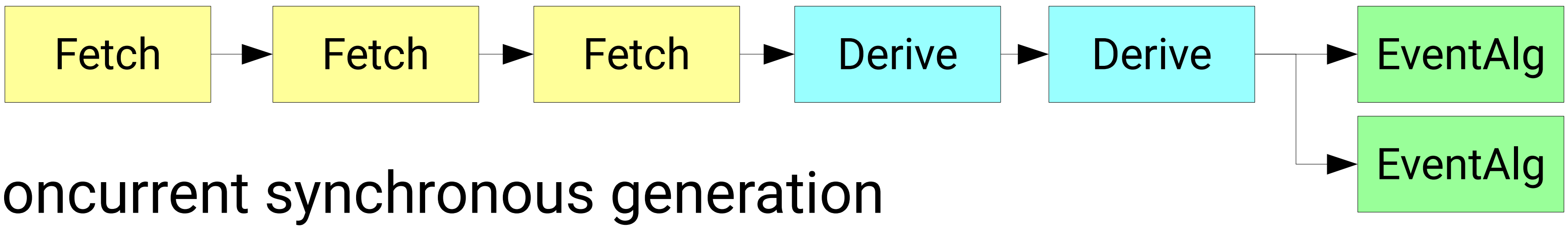
- **Detector state** is used during event processing
- It can be decomposed into sub-components
- Some of these are **time-dependent**
  - Time evolution modeled as sudden changes
  - Versioned data with Intervals of Validity (**IoV**)
  - We call a version of a state component a **condition**
- Must load/compute conditions as needed

When to generate?

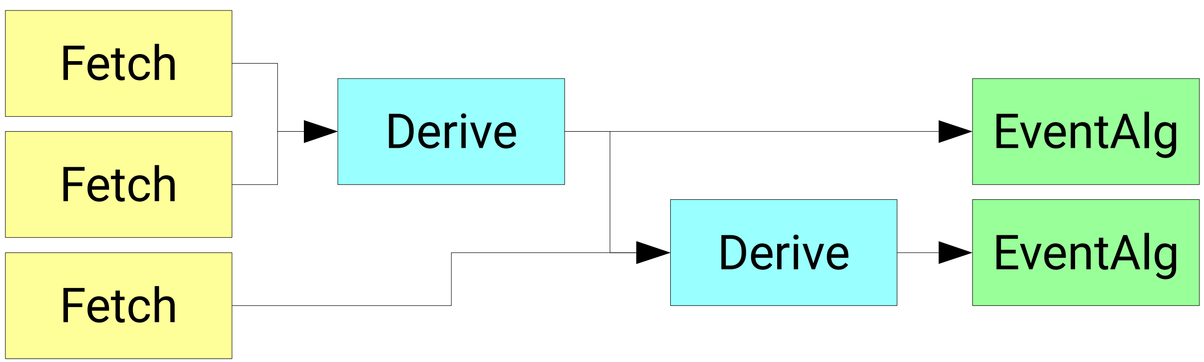
- Important design choices to be made here!
- Prepare everything **before** event is processed?
- Perform async writes **during** event processing?

A visual comparison

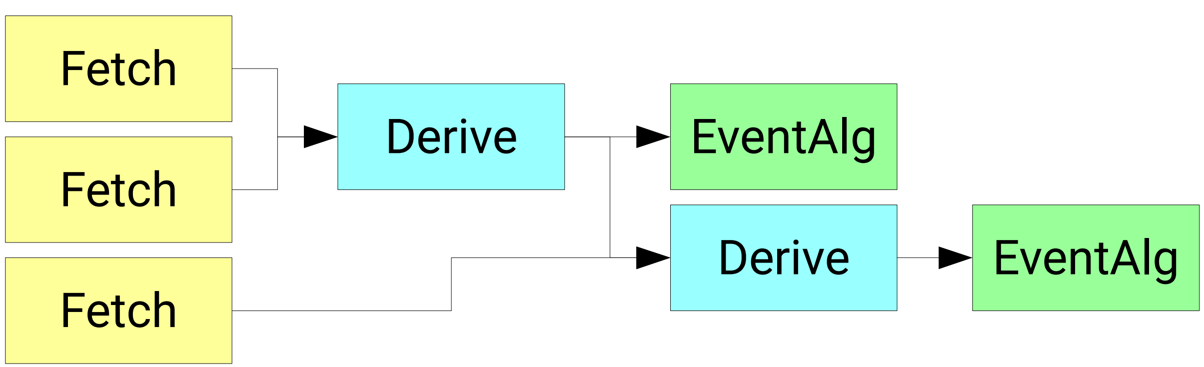
- Sequential synchronous generation



- Concurrent synchronous generation



- Asynchronous generation (overlapping w/ first event in IoV)



When to generate?

- Important design choices to be made here!
- Prepare everything **before** event is processed?
  - Eases progressive migration
  - Condition reads do not require synchronization*
  - Simplifies the design in many ways
  - Drawback: Scheduling of new events is delayed
    - Concurrency can reduce the delay, not eliminate it
    - An issue if condition switches very expensive/frequent

* Given careful storage design: no centralized, growable, blocking container...

When to generate?

- Important design choices to be made here!
- Prepare everything **before** event is processed?
- Perform async writes **during** event processing?
  - Reduces event scheduling delay, at a price
    - More complex design and code migration
    - Coupling condition & event processing is unfortunate
    - Requires blocking-aware (more complex) scheduling
    - Later events with same conditions still need to wait
  - This is the road that ATLAS have taken so far[4]

[4] C. Leggett, "Managing Asynchronous Data in ATLAS' Concurrent Framework", 2016-08-05

When to generate?

- Important design choices to be made here!
- Prepare everything **before** event is processed?
- Perform async writes **during** event processing?
- Can we support both approaches? Should we?

Generation process

- Need a configurable, plugin-based architecture to account for experiment specifics
- Must separate “raw” IO and “derived” compute
  - TBB tasks are optimal for CPU-bound workloads
  - IO starves CPU unless extra threads are added
- Concurrent generation[5] is a nice optimization!
  - Can TBB handle a hybrid task/thread flow graph?
  - Integration in the existing Gaudi architecture?

[5] C. Leggett, “Conditions Access”, 2016-01-20

Performance requirements?

- Condition access must be fast (many/event)
  - Reentrant handles have a cost
  - Must allow caching with a reasonable lifetime
- Generation: depends on experiment choices
  - IoV switching rates (global, individual)
  - Costs of condition vs event processing
- Better designs possible when generation is rare
  - Less coupling, more efficient access and GC

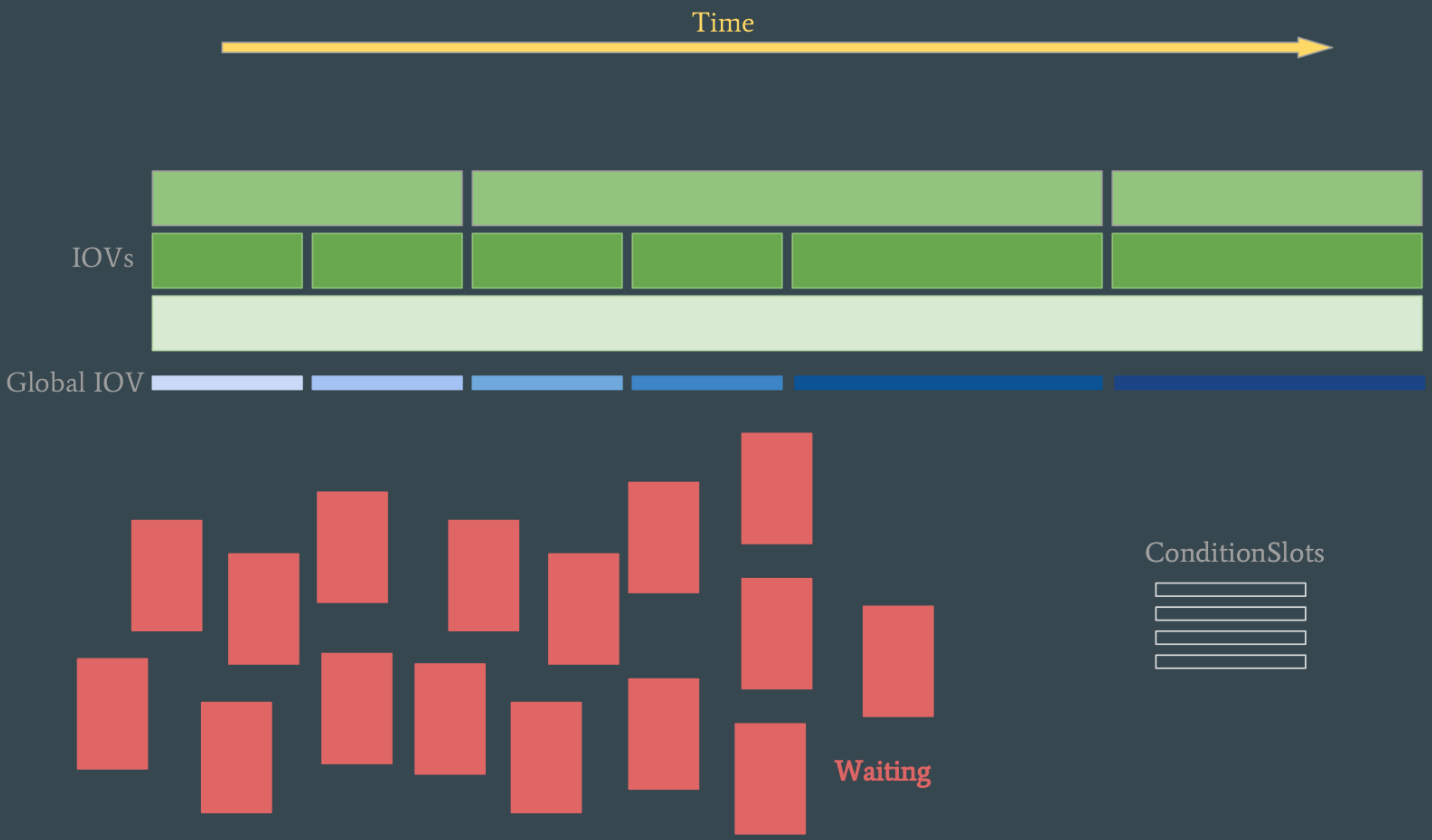
Condition client interface

10000 foot view[6]

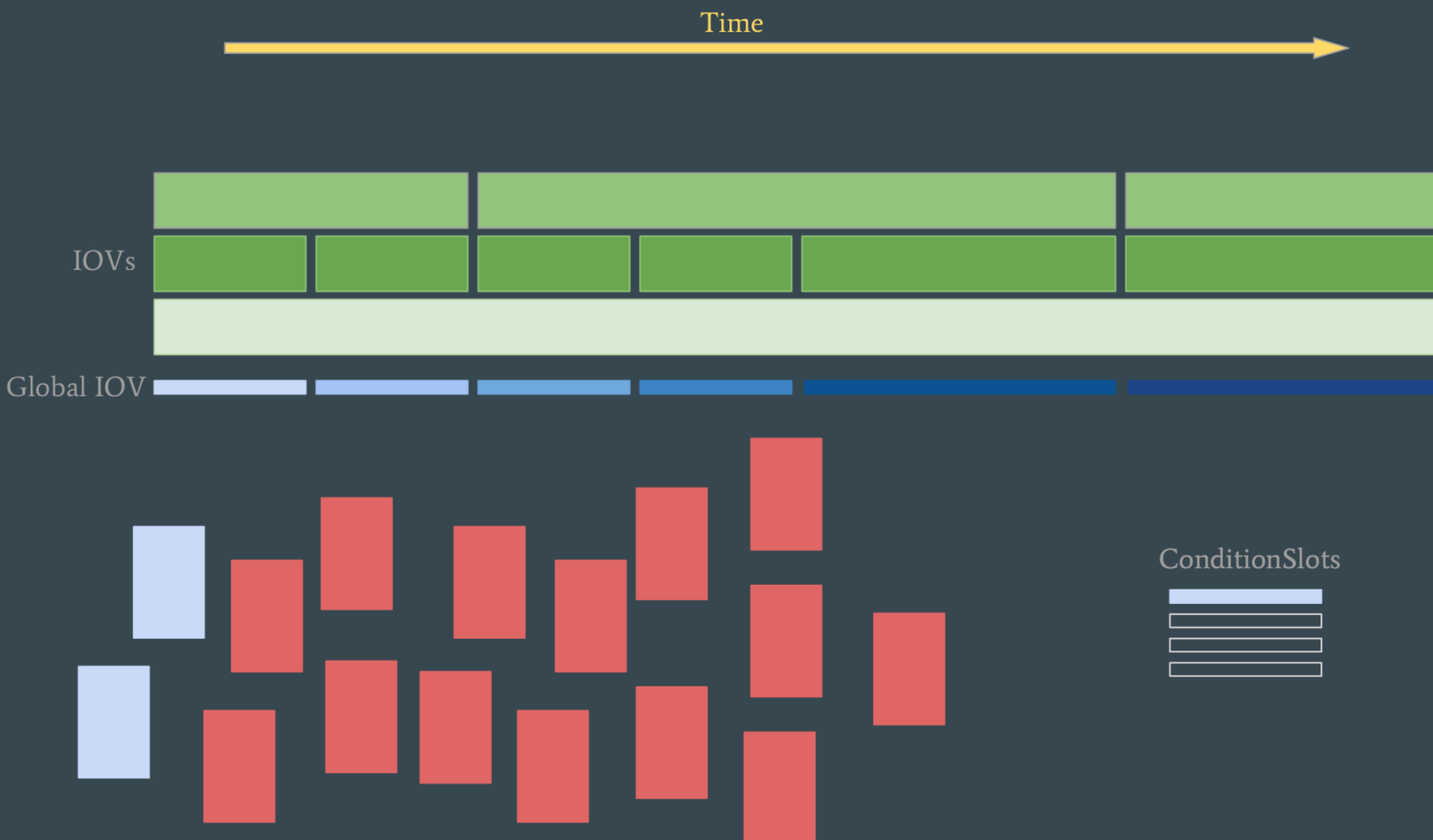
- Every time a new event comes up
  - EventLoopMgr calls the **ConditionSvc**
  - Provides the **time point** associated with the event
  - Requests a **ConditionSlot**
- ConditionSvc retrieves or constructs such a slot
- (Smart-)reference to it stored in EventContext
- Used by **ConditionHandles** to access data

[6] Iterated from B. Hegner, "Condition and Run Handling in Gaudi Mila", 2016-08-08

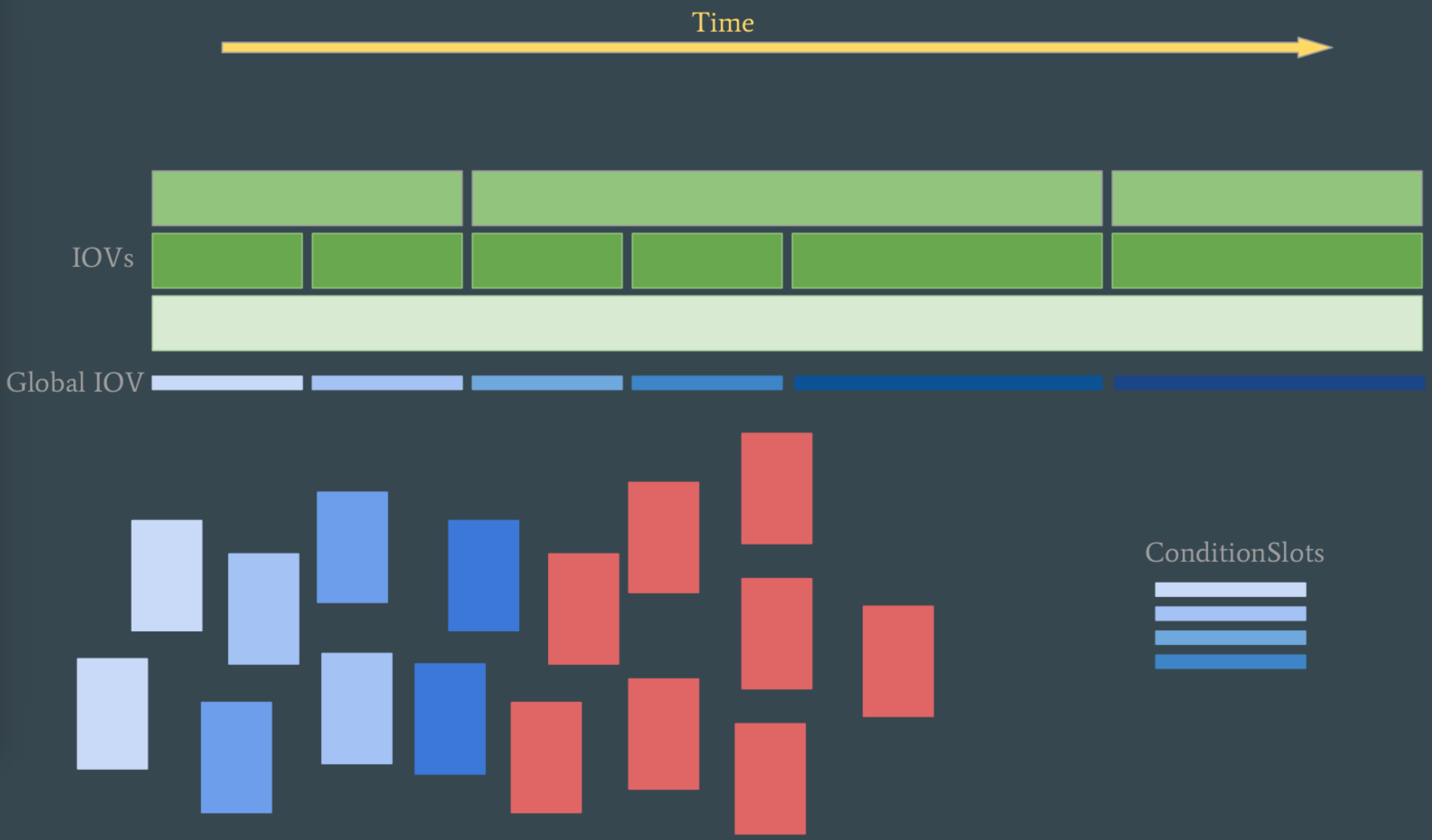
Let's visualize this[6]



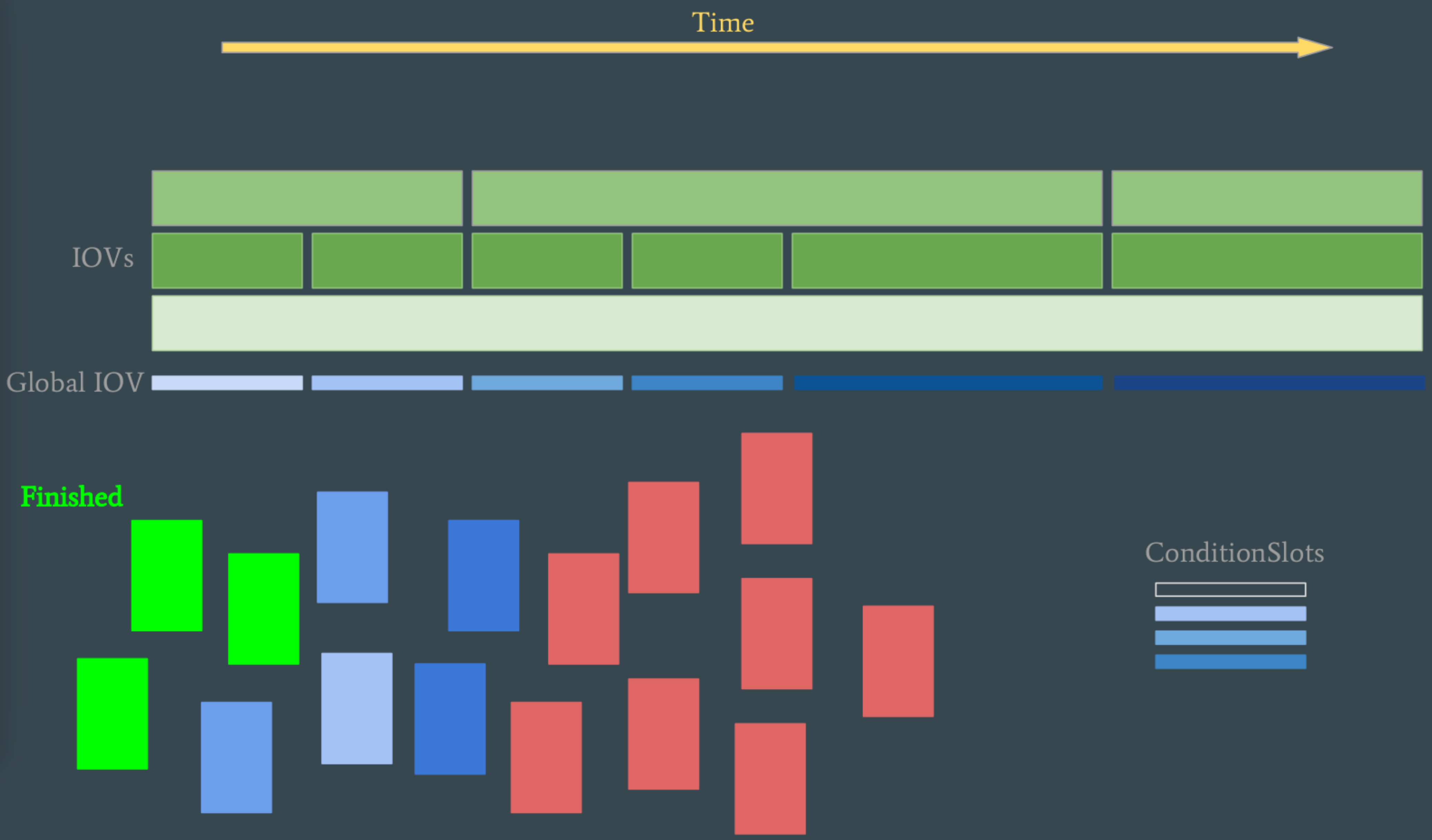
Let's visualize this[6]



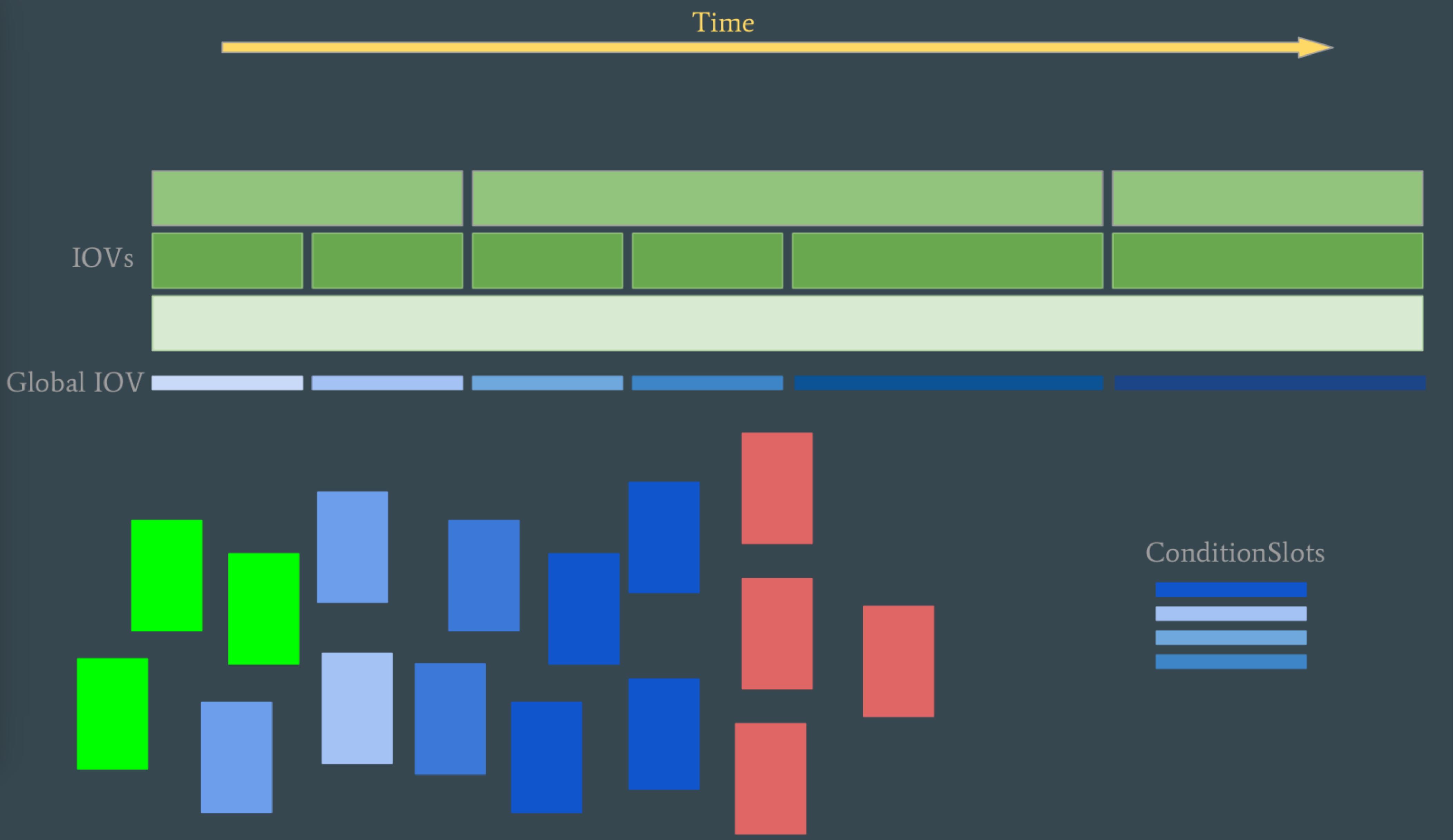
Let's visualize this[6]



Let's visualize this[6]



Let's visualize this[6]



ITimePoint

- Common interface to experiment time points
  - Could & should be a template argument concept
  - Should not be used all over the place then
- Inspired by ATLAS' EventID design
  - Without experiment-specific data format
  - Refocused on timing, not events
  - Can also represent condition IoV boundaries

ITimePoint concept

```
bool is_comparable_with( const ITimePoint & other ) const

bool operator<( const ITimePoint & other ) const
…
class InvalidTimePointComparison : public std::exception { … }

const size_t representations_count = // Experiment specific
using Representations = std::array< ITimePoint,
                                    representations_count >
Representations split_representations() const
```

TimeInterval

- Framework-level notion of an IoV
- Based on pairs of ITimePoint
- Can be intersected to produce global IoVs

TimeInterval interface

```
TimeInterval( const ITimePoint & start,
              const ITimePoint & stop )

bool contains( const ITimePoint & what ) const

void intersect_with( const TimeInterval & source )
```

ConditionSlot

- Abstraction: Complete set of conditions
  - Makes access & refcounting easier, more efficient
  - Can share data with other ConditionSlots
  - User can bound amount of slots in flight
- Has a global IoV = intersection of child IoVs
- Design concept, not directly exposed

IConditionSvc

- Interface to the condition management system
- Given the timing data for a new event...
  - Look for a matching busy slot in registry
    - If there is one, increment refcount and return
  - If no slot matches, try to allocate and fill up one
    - Reuse condition data from neighbouring slots
    - Generate the rest (do it synchronously or schedule it)
  - If no free slot left, ask EventLoopMgr to retry later

IConditionSvc interface

```
// Concrete ConditionSvc must provide this constructor
IConditionSvc( size_t slot_amount )

// Used when initializing algorithms, registers condition needs
template< typename RefToConst >
ConditionHandle<RefToConst> make_handle(
   const ConditionKey & key
)

// Used when a new event comes in
StatusCode prepare_slot(       ConditionSlotID & target,
                         const ITimePoint      & time )

// Called after event processing, possibly by ~EventContext()
void release_slot( const ConditionSlotID )
```

ConditionHandle

- Used by Algorithms to read conditions
- Under the hood, goes through ConditionSlot
  - Reentrant by design (current slot in EventContext)
  - Should not be accessed in a loop
    - Client can safely cache references during execute()
    - Slot lifetime is managed on the scale of entire events

ConditionHandle interface

```
template< typename RefToConst >
class ConditionHandle {

   // ConditionHandle are created by the ConditionSvc
   ConditionHandle() = delete

   // Fetches a const-reference to the condition data for the
   // associated condition key, in the active condition slot
   RefToConst get_current() const

}
```

Garbage collection design

- Condition slots are reference-counted
  - Increment when new event is started
  - Decrement when event processing completes
  - Extremely efficient in the common case!
- Individual conditions are also refcounted
  - Measure amount of **slots** which share the data
  - Some overhead on slot creation/deletion

What's next

- So far, focused on condition access
  - Decoupling clients from condition management
- Also need to generate and store conditions
  - Start with a sequential generation model
  - Request the metadata needed for concurrency & IO
    - Is a given condition generator thread-safe?
    - Can it block the underlying OS thread?
  - Provide a unified interface to storage backends

Questions and comments!

A place to hook

- Must be able to tell about incoming events…
  - ...and the time when they occurred
- The best we have now is beginEvent
  - No notion of event time
  - Incidents are deprecated
- A use case for the replacement of incidents?

Detector description toolkit

- Conditions could be managed by the detector description toolkit (e.g. DD4Hep) and merely linked into event data storage
  - Still need to think about **toolkit** requirements
  - Must allow concurrent storage or serialization
  - Interface must allow garbage collection
  - May require a notion of time as well
  - Must make linking for every event efficient

Give everything an IoV?

- Event = point-like IoV, global = infinite IoV?
  - Would cause lots of unnecessary IoV checks
  - Would force unnecessary scheduler complexity
  - IoV management relies on experiment specifics, don't want to put these everywhere