Design to Simulate 15 Minutes of Running

David Jaffe & Brett Viren

Physics Department

BROOKHAVEN NATIONAL LABORATORY

Daya Bay Simulation Workshop

Overview

1. Design Goals and Overview
2. Stages of Processing
3. Nonlinear Processing
4. The Design
5. Modes of operation
   - Steady State Generation
   - Partial Generation
   - Intermediate Files
6. Notes on Implementation
7. Summary
8. Meng's Proposal

Design Goals and Overview

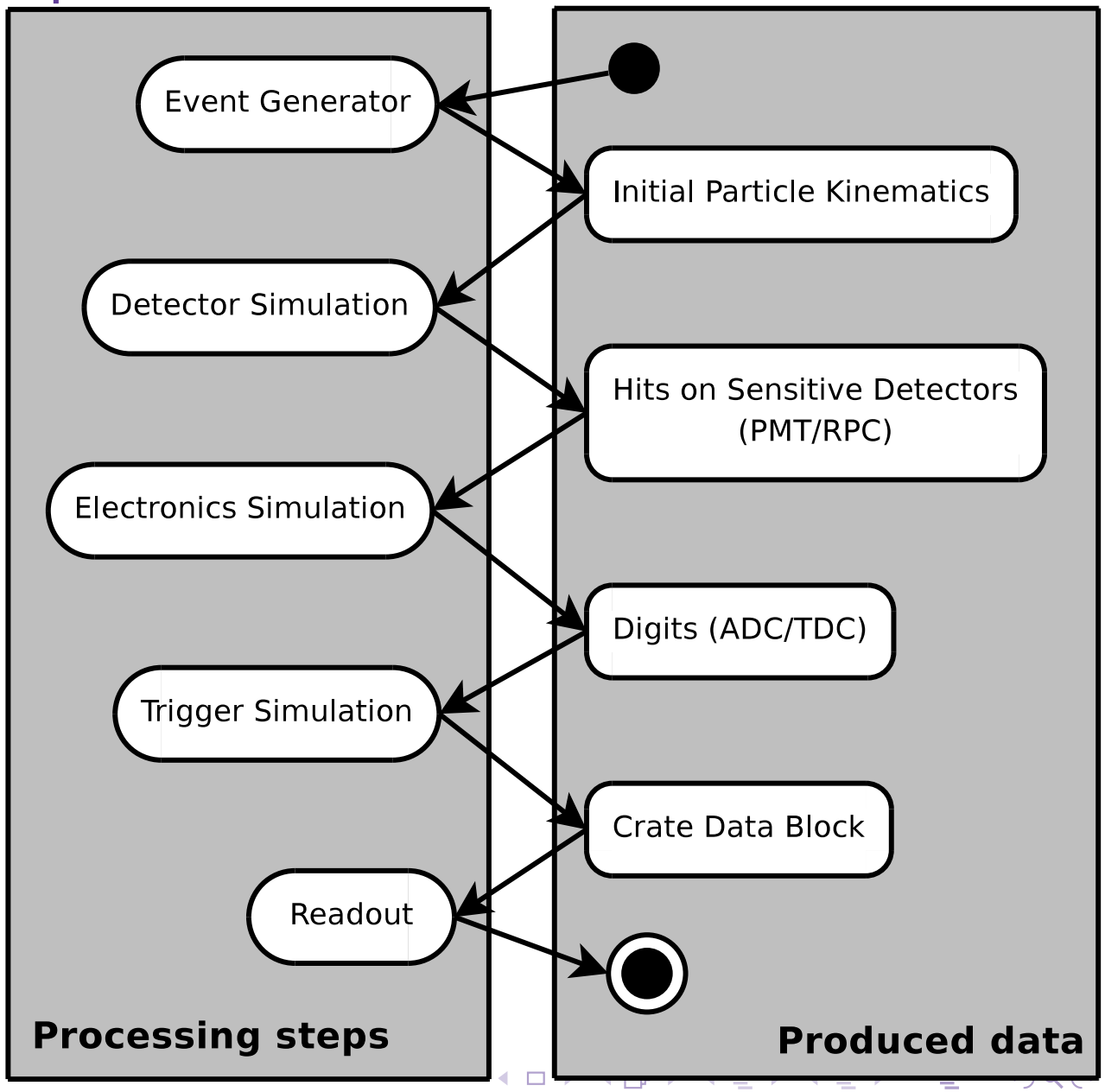
Independently, Steve Kettell and Jun Cao recognized the usefulness of generating a sample of "real" or "cocktail" MC data with all event types properly represented.

This design shows how we might produce this MC data. It also has the additional features:

- Although 15 minutes is the target, there is no limit on the duration of time to be simulated.
- It modularizes the processing so any part can be tested independently.
- It allows for the types of events for consideration to be configurable.
- It supports different modes of operation including saving intermediate results to file for subsequent read back.

Processing stages, simple view

1. Event Generator produces Initial Particle Kinematics
2. Detector Sim. produces Hits
3. Elec. Sim. makes Digits
4. Trig. Sim. forms Crate Data Blocks
5. Final readout

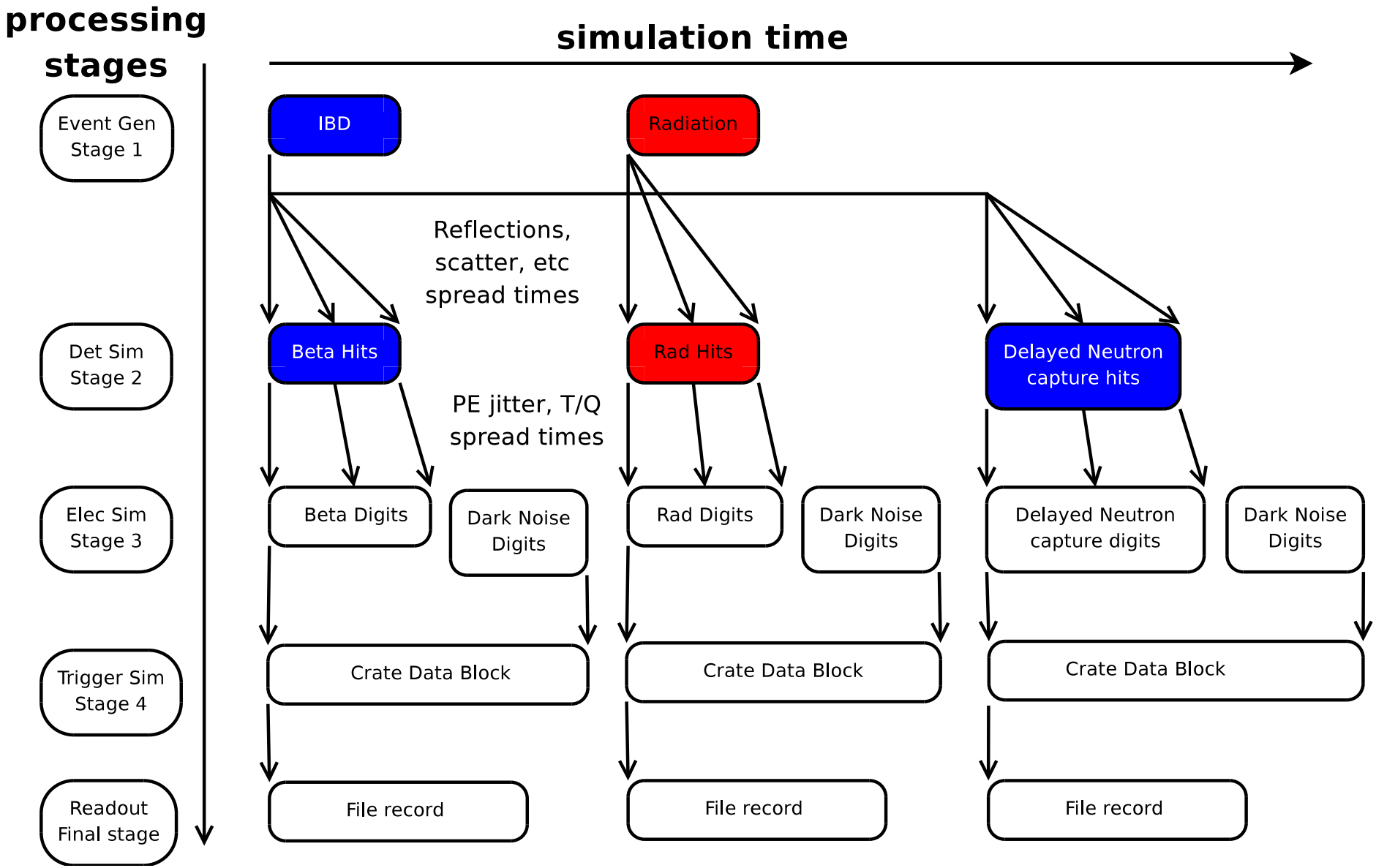


Nonlinearities in processing

Real world: events are asynchronous, overlapping and spread out in time. Processing is not always so linear.

- The U/Th radiation generator produces a decay chain which spreads initial kinematics through time.
- Inverse beta decay can produce prompt and delayed hits in the detector sim.
- Optical photons are spread in time due to extended production source, scattering, scint. and reflections.
- PMTs have PE transit time jitter as well as random, uncorrelated dark noise.

Example: Inverse β Decay with Radiation



Example: IBD + Rad, continued

(Skip during presentation, for reference only)

From the previous figure.

- An Inverse β-decay initial kinematics are fed to the det. sim.
- The det. sim. produces prompt and delayed hits.
- At this point the process has the full results of the det. sim. available.
- Next in line is a radiation that comes in between the prompt and delayed hits. What to do?
  - Process IBD hits $\rightarrow$ digits $\rightarrow$ crate data and write out to file. Oops, now we have a Radiation event and the file is out of order
  - Okay, hold on to the IBD hits "long enough" to assure all other kinematics that might lead to overlapped hits have been processed. What is "long enough"? It depends....
- What about dark noise? If IBD is fully processed out to file, where to we insert dark noise?

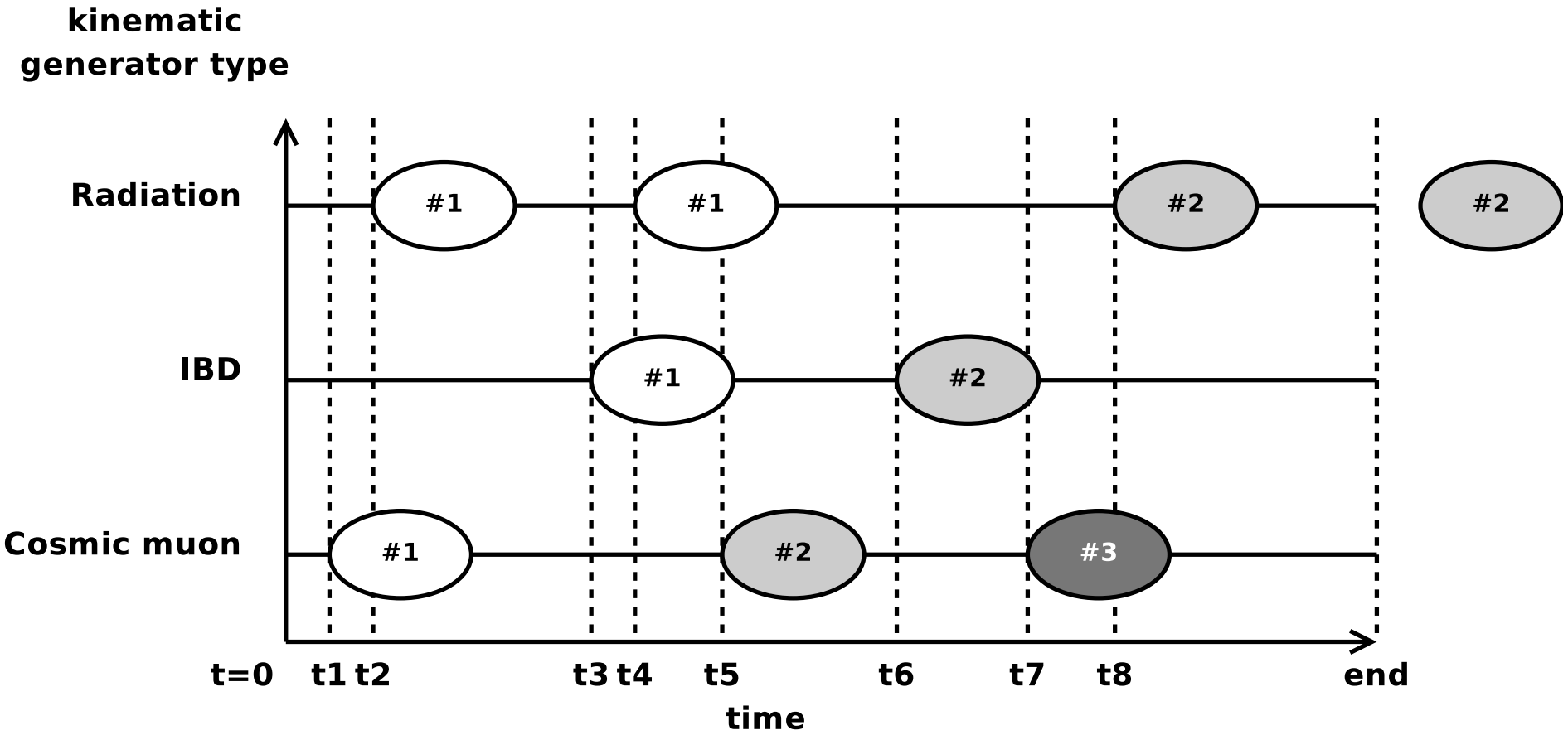
Very confusing situation!

Elements of the design

Abstract the problem to tame the confusion! Elements of the design:

- The job consists of a series of processing **stage** objects.
- Each stage provides output **data elements** of a specific data type which are produced by one or more **processors** managed by the stage.
- Processors produce the data elements for the stage, potentially using input data from the prior stage. When a prior stage's data element is used it is considered **referenced** by the produced output data.
- Each stage presents its data as a **time ordered collection** of elements through an **iterator** object.
- Stage data must be retained until it is no longer referenced.
- A **time** is given to each process when data is requested. The process should return new data only if this time is equal to or greater than the time of the data the processor last returned.

Example: Initial particle kinematics generation



Stage 1, generation of initial particle kinematics with three event types.

Example: Kinematics, cont.

(Skip during presentation, for reference only)

In the previous diagram:

1. Iteration starts at $t = 0$ and all process return new data to the stage since they have not ever returned prior data.
2. The stage time-orders the results and returns Cosmic-μ #1 at $t = t_1$. This now the "current time".
3. Next iteration, all processors are asked for new data at or after $t = t_1$.
4. Cosmic-μ generates #2 at $t = t_5$, the others add no new data.
5. Stage time-orders the data and returns Radiation #1 at $t = t_2$.
6. etc.

Different modes of operating

This design supports various different modes of operation:

Steady state generation with no pre-defined time limit.

Partial generation of only a subset of all stages.

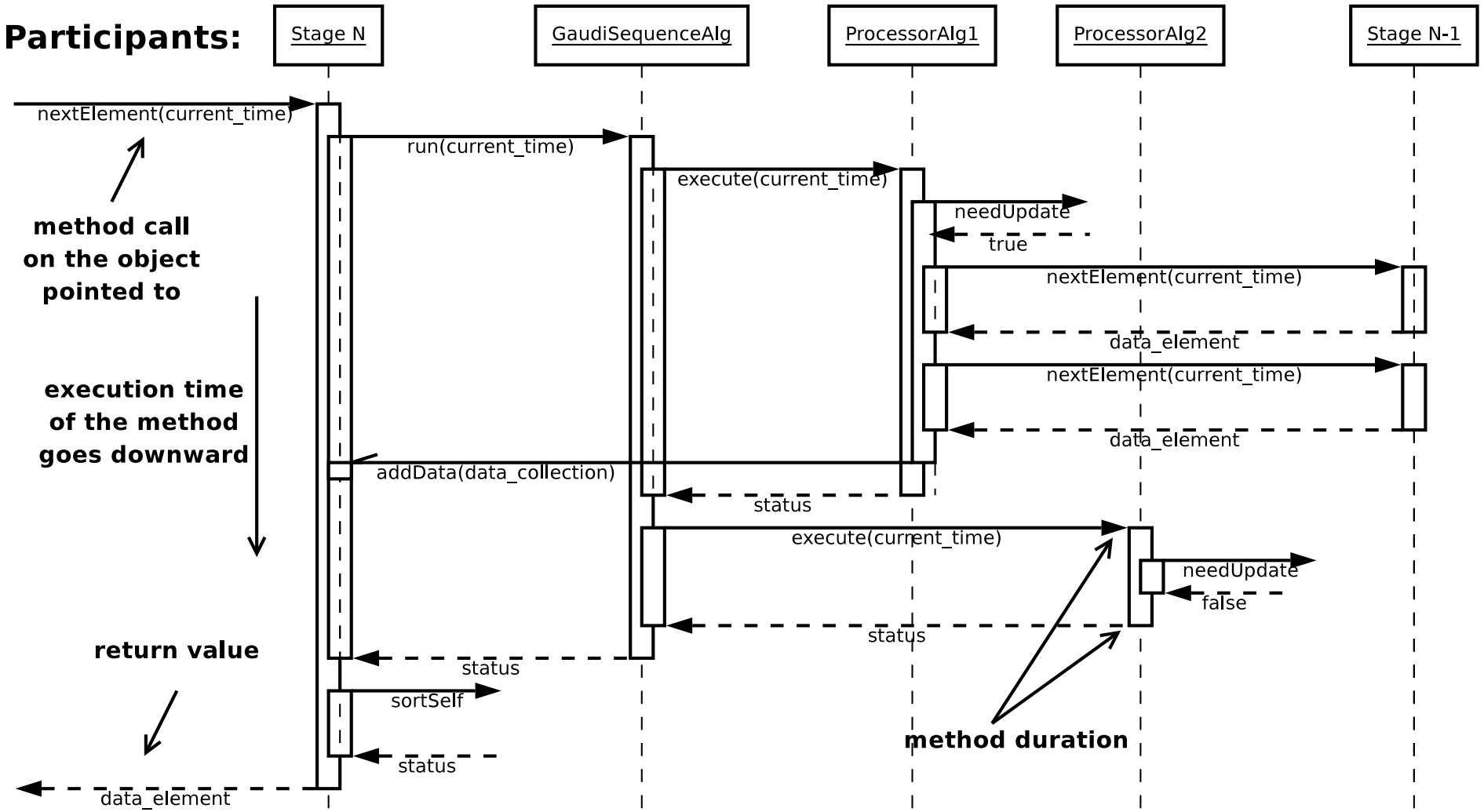
File I/O used to store intermediate stage output and later read back in to mimic generation.

Each mode is described in detail.

Steady state generation

- It means: start the simulation and let it run as long as you need.
- Directly analogous to real data taking (ignoring start-of-run effects).
- No fixed simulation duration ⇒
  - No a'priori estimate of buffering needs, can not store full run in memory.
- Requires "just in time" processing ⇒
  - Processors may need to delay output data request in order to access a prior stage's data as input to its processing.
  - Explicit support in Gaudi for this type of operation.
- To drive the simulation, simply read data from the last stage.

Just in time processing



A "UML Sequence Diagram" showing interaction between Stage N and Stage N-1.

Just in time processing, continued

(Skip during presentation, for reference only)

In the previous diagram:

1. A iterator (not shown) starts to loop over Stage N's data and calls its `nextElement(current_time)` method with the iterator's current time.
2. The stage queries all processors for data at or after `current_time`.
   - `ProcessorAlg1` needs an update and needs to ask Stage N-1 for data. It returns new data.
   - `ProcessorAlg2`'s prior data is still older than `current_time` and adds no new data.
3. All processors return and Stage N sorts the results and returns the most recent data element.

Partial Generation

If one is only interested in hits, there is no reason to simulation electronics and triggers, just loop over the Detector Simulation stage.

Intermediate Files

The above assumes everything is generated. But, we may want intermediate files:

- A particular stage (eg. det. sim.) may be very time consuming, and you want to re-use its results while developing subsequent stages.
- To develop a stage's processor in isolation from running other stage code.
- To study intermediate results after the processing has finished.
- To plug in results generated in some unrelated, external software.

Easy to support: write replacement stage object that populates the Transient Event Store with data from file instead of generating it.

Map of Design Object to Gaudi Features

This design maps very well to features Gaudi supplies.

Design Element	Gaudi Feature
Data Elements	Daya Bay Data Model
Stage	Gaudi Tool
Stage Data Collection	Transient Event Store
Processor	Gaudi Algorithm
Collection of processors	GaudiSequenceAlg
Detector Simulation processor	GiGa (Gaudi/Geant4 interface)
References	Gaudi built-in reference counting support

Summary

"15 Minutes of Fame"[1] design from DocDB#1532

- Modular, lets one plug in different stages and their processors
- Flexible, lets one simulate in a steady state manner, write/read intermediate files or focus on just one or more stages.
- Continuous simulation possible, no need to estimate numbers of events.
- Well matched to a Gaudi based implementation.

[1] Blame David for the name!!

Meng's Proposal

Meng presents (DocDB#1564) an idea that is centered on only using intermediate files. It is somewhat complimentary to our (D&B) design.

In summary (apologies for any mistakes):

- Driven by the desire to support changing one or more event rates w/out needing to re-simulate
- For each event type:
  - Use a nominal rate, simulate enough to cover potential changes
  - Simulate all the way: kine→hits→digits
  - Write results to file, one per event type.
- Read back in all files, drawing events based on specific desired rates, to properly "mix" the data.

Please also see DocDB#1564 for very useful tables of rate estimates and other timing quantities!

Meng's Proposal, cont. Some comments.

Comments on Meng's Proposal:

- Not clear to me if it is easier to implement than the D&B design.
- Still need something like **stages** (one anyways) and their **processors**.
- Need to read in new data from each file only at the proper time; assign one processor per file.
- Need to post-process data to apply proper time offsets.
- How does one properly place events that happen to occur between digits from delayed coincidence? (eg, IBD + Radiation example figure)
  - Still need "just-in-time" type pattern?
  - First over-fill store with delayed coincidence digits? How to keep user code away from the delayed IBD digits until the Radiation ones are used?