Drivig random numbers in art

Gianluca Petrillo

LArSoft stakeholders' and partners' meeting, April 22nd , 2014

Random number generators in art

art provides a central manager of random generators: the RandomGeneratorService (RGS) service, managing engines from CLHEP library.

- an *engine* can produce a sequence of pseudo-random numbers
- RGS assigns engines to the modules
- each engine is used only by one module
- one module can use more engines

Each module asks *all* the engines it needs in its constructor, specifying a seed and an optional label:

```
createEngine(seed); // default random engine
createEngine(seed_OptRandom, "OptRandom"); // another one
```

When the module needs a generator, it asks RGS for the engine:

```
art::ServiceHandle<art::RandomNumberGenerator> rng;
CLHEP::HepRandomEngine& engine = rng->getEngine(); // or ("OptRandom")
CLHEP::RandFlat flat(engine); // example: extract with flat distribution
```

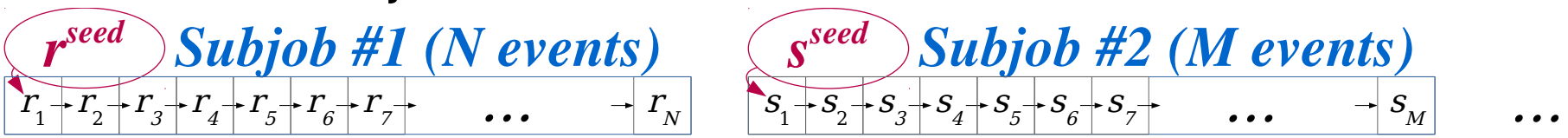
Continuing a previous job

Use cases:

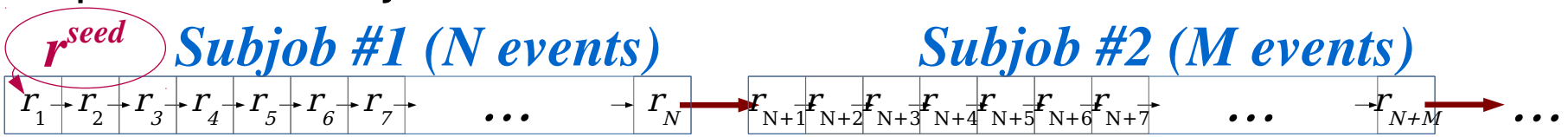
- split a long job in a sequence of subjobs, run the first subjob, and have the second subjob run from where the first ended
- extend an existing job with additional events

... using only one sequence of random numbers across the jobs rather than one different sequence for each job.

Uncorrelated subjobs:



Sequence of subjobs:



How to continue random sequences across jobs

`RGS` can restore the status of all the engines at the beginning of a job, and/or save them at the end of the job.

First job:

```
services.RandomNumberGenerator.saveTo:   "FinalStateJob1.txt"
```

Second job:

```
services.RandomNumberGenerator.readFrom: "FinalStateJob1.txt"
services.RandomNumberGenerator.saveTo:   "FinalStateJob2.txt"
```

Note that the `saveTo` file must be copied back from the job, and shipped to the next one.

This will preserve *all* the random engines known by `RandomGeneratorService`.

Reproducing an existing job

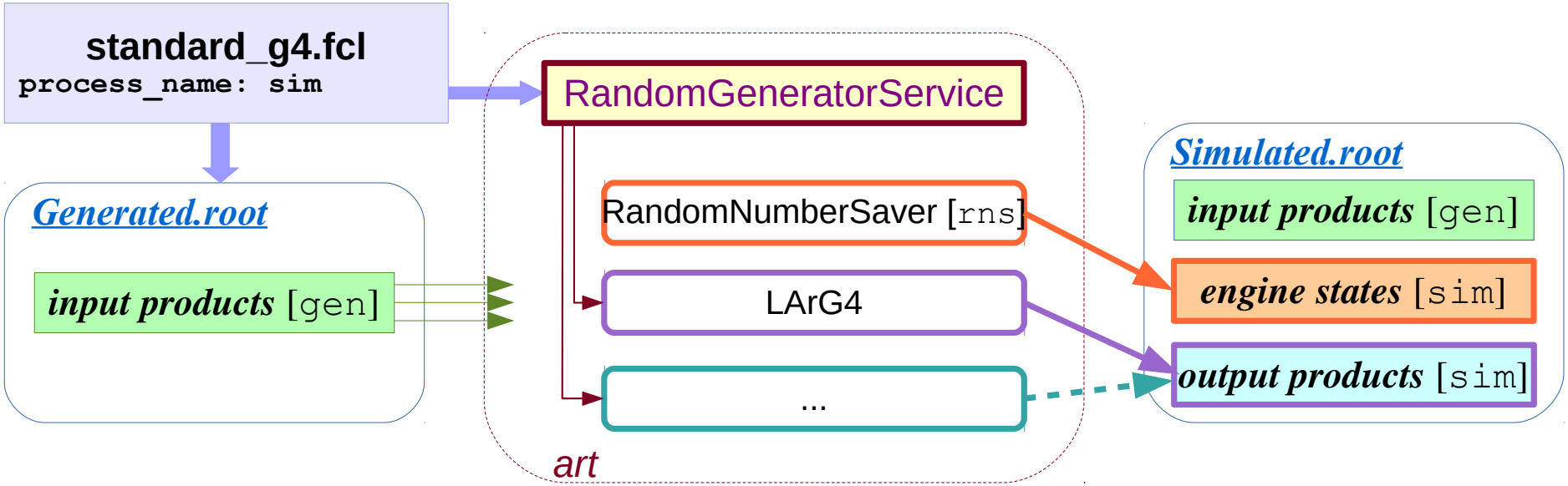
Use cases:

- a code change: we want to compare products of old and new code
- a job crashes: we want to reproduce the crash, ideally jumping directly to the troublesome event

`RandomGeneratorService` can also read the state of all engines from the input file, and reseed them event by event.
The `RandomNumberSaver` module can save the current state of all engines as a product into the event.

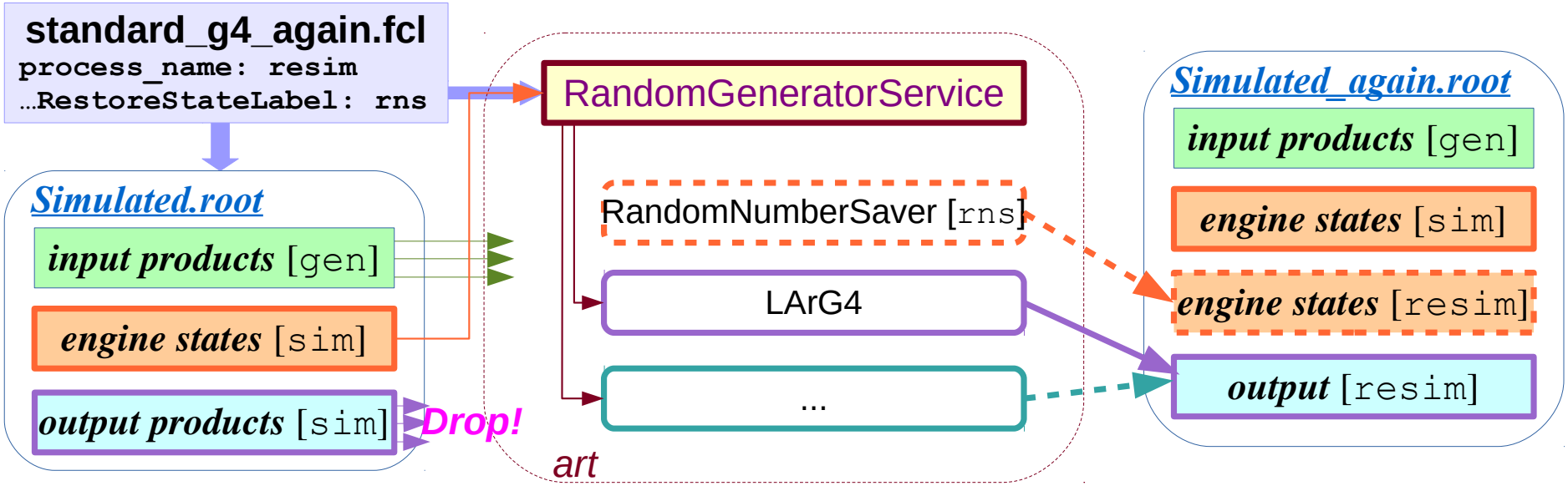
How to produce a job

```
lar -c standard_g4.fcl -s generated.root:
```



How to reproduce the same job

`lar -c standard_g4_again.fcl -s simulated.root:`



The new input file could be something like:

```
#include "standard_g4.fcl"

process_name: resim
services.RandomNumberGenerator.restoreFromLabel: "art::RNGsnapshots_rns__s
source.inputCommands: [ "keep␣*", "drop␣*_*_*_sim", "keep␣*_rns_*_sim" ]
```

where the `restoreFromLabel` will be used by a `art::Event::getByLabel()` call to read the engine state product.

Central control of the seeds

Use cases:

- manage all the job seeds with minimal bookkeeping and effort
- make sure no jobs have the same seed
- be able to rerun a job

No code ready in `art`.

mu2e has written a service with the purpose to address the two cases described above.

mu2e `SeedService` service

- `SeedService` (SS) provides a seed to each module asking one (or more!)
- the seeds are determined by:
  1. a base seed (`baseSeed`)
  2. the order of the request (but it depends on policy)
- a few different policies are already implemented; the simplest: the first seed is `baseSeed`, the second seed is `baseSeed + 1` etc.

Each engine should be immediately seeded on creation in its module constructor; for example:

```
createEngine(pset.get<int>
  ("Seed", art::ServiceHandle<mu2e::SeedService>()->getSeed()));
createEngine(pset.get<int>("Seed",
  art::ServiceHandle<mu2e::SeedService>()->getSeed("OptRandom")),
  "OptRandom");
```

The base seed is set in the configuration file, and can also printed in the `Info` output stream of each job.

`SeedService` service vs. `RandomNumberSaver`

Respect to the solution with `RandomNumberSaver` (RNS):

- SS does not allow to jump to a specific event
- to reproduce an exitsing run, SS relies on the configuration to be exactly the same
- RNS does not provide any control at all on the seed
- SS requires each single module to explicitly use its services

Example: mu2e approach

- split a production job in N jobs (each with an index $i_{\text{job}} \in [1, N]$)
- initialize `services.SeedService.baseSeed` with i_{job}

With the proper policy ("linearMapping") this is enough to guarantee unique sequences for the whole production.

Other use cases?

- the available software within `art` addresses needs from a couple of use cases
- the `SeedService` could address another one
- are there comments or opinions about this last one?
- are there requests for other use cases?

Unusual seeding practises

`ShowerSelectorFilter` and others always use a random seed

`fuzzyCluster` and `HoughLineFinder` use either a random seed, or the same random numbers for all the events (reseeding each event with the same seed)

... and a few more

Recommended:

```
createEngine(pset.get<int>
  ("Seed", SeedCreator::CreateRandomNumberSeed()));
```

Further reading

Documentation from mu2e

Random generator service: http://mu2e.fnal.gov/public/hep/computing/Random.shtml

Other `art` services: http://mu2e.fnal.gov/public/hep/computing/artNativeServices.shtml