BackTracking Overhaul in LArSoft

J. Stock & J. Reichenbacher
South Dakota School of Mines and Technology.

The BackTracker and PhotonBackTracker.

BackTracker

- Rebuild is run whenever the BackTracker or PhotonBackTracker is configured.
- The method of data return from BackTracker functions is extremely inconsistent.
- BackTracker functions names are often incorrect or misleading. (ID and IDE are regularly conflated in the current naming).
- Extra modules are required to run the backtracker at the right time if it can't run correctly at the start of an event. User feedback leads me to believe this usage model is confusing and difficult for end users.

PhotonBackTracker

- The PhotonBackTracker follows the BackTracker model as closely as possible, and so has all of the same limitations and issues.
- The PhotonBackTracker competes for resources with that BackTracker, causing undefined behavior with the ParticleLists from an event.
- In the PhotonSimulation, important physics is done during the detsim stage, which the PhotonBackTracker is blind to.
- The current model for the PhotonBackTracker does not know that multiple optical channels exist per detector.

Rebuild.

BackTracker

- The rebuild stage causes a try/catch for every event during the generation stage, and the LArG4 stage.
    - This TryCatch also has an output to the log for every event.
- The rebuild stage causes unnecessary memory usage by recalling data product from the event regardless of whether or not they are used.
- The Rebuild Stage of BackTracker calls the particle list from the event, and then defines an EveIdCalculator to use with that list.

PhotonBackTracker

- The rebuild stage causes a try/catch for every event during the generation stage, and the LArG4 stage.
    - This TryCatch also has an output to the log for every event.
- The rebuild stage causes unnecessary memory usage by recalling data product from the event regardless of whether or not they are used.
- The Rebuild Stage of BackTracker calls the particle list from the event, and then defines an EveIdCalculator to use with that list.

Because both services define the EveIdCalculator for the event's ParticleList, whichever happens to be initialized second will be the EveIdCalculator used. This is an undefined behavior.

The ParticleInventory service.

- One service to handle all ParticleLists for backtracking purposes.
- One service to handle the bulk of the rebuild phase, reducing duplication of effort between the BackTracker and PhotonBackTracker, and eliminating it for the ParticleLists.
- Lazy Rebuilding can be implemented, to prevent unnecessary rebuild steps from being run, and eliminating many predictable and unnecessary log warnings.

All functions that are being factored out of the backtracker will have copies in the BackTracker calling the new service, and printing a log warning instructing the user to make the call from the new service instead. This functionality can be left in LArSoft as long as needed for all users to update their code to the new service, reducing the impact of these breaking changes.

The particle inventory service does not require any experiment specific configuration. It can be easily and quickly added by each experiment to their configured services as a copy of the standard service.

Inconsistent Functions and incorrect names.

```
void ChannelToTrackIDEs
(std::vector<sim::TrackIDE>& trackIDEs, …)

std::vector<sim::IDE> TrackIDToSimIDE(int
const& id)

const simb:MCParticle* TrackIDToParticle(int
const& id) const

std::vector<sim::TrackIDE> HitToTrackID(...)
```

As the examples to the left show, there is no real consistent form to how the various backtracker functions return information to the user.

The last example shows a more significant issue, where the name of the function clearly implies one output, while the function actually returns something quite different.

I propose making all functions as they currently stand available to the user, and making new functions using the name FunctionPtr and FunctionCp to explicitly pass the requested object either as pointers (art::Ptrs where possible, c pointers where not) or as copies, allowing the user to determine the best method for their specific use case.
Some object must be passed as copies because they do not exist in the data products themselves (TrackIDEs are one such case), though there are very few such cases.

Each of these issues is similarly found in the PhotonBackTracker, and the solutions recommended are the same.

The PhotonBackTracker is blind to DetSim.

PhotonBackTracker information is stored as

Int (op det #) , vector<pair<double (time) , vector<SDP (TrackID, nPhotons, energy, xpos, ypos, zpos) > > >

This is based on the SimChannels objects, to allow as much consistency between the modules as possible. For the DUNE use case, we need to be able to store channel specific backtracking information as well. This would be added to the event during detsim.

The current data product (OpDetBackTrackerRecords) can be expanded to allow for storing channel specific information (We anticipate minimal user impact as the PhotonBackTracker currently has limited adoption outside of DUNE PD-Sim/Reco).

A new data product can be made to track how signals are detected by individual channels in parallel with the OpDetBackTrackerRecords. (Requires a OneToMany art::Assn between the new records and the existing records.

Status

A ParticleInventory service is ready to implement. It is just waiting on the updates to the BackTracker and Photon BackTracker so that I may begin rigorous CI testing.

BackTracker updates are over 50% completed.

PhotonBackTracker updates will hopefully begin this week.

I am trying to finish all updates to the BackTracking software this month, so that I will be here for the first couple weeks of use to assist with any issues that may arise.

Questions?