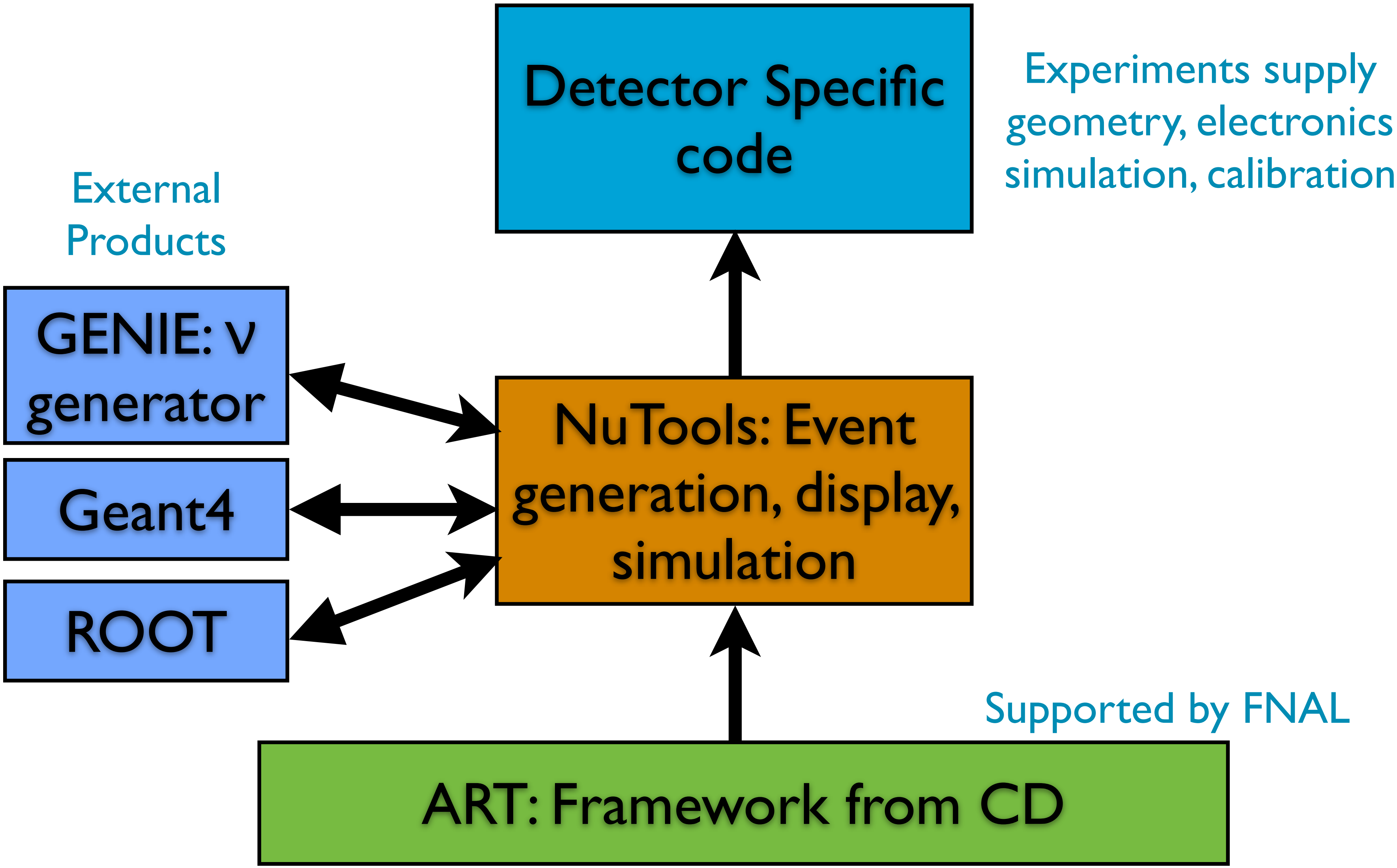
Using NuTools for DUNE ND Simulation

Brian Rebel
October, 2015

What is NuTools?

- NuTools is a set of C++ packages designed to be generically useful for neutrino experiments
- It assumes that an experiment is using the ART framework in general
- It provides
    - Interfaces to popular simulation code like GENIE, G4, CRY
    - Basic event display functionality to be used with ART
    - Tools to interact with the GENIE reweighting functionality, beam reweighting as well
    - Definitions of simulation data products to characterize particles, collections of particles for a given trigger, neutrino flux, etc
- Currently it is used by NOvA and LArSoft (and all experiments using LArSoft)
- Benefit of using NuTools is that it is constantly checked by several experiments and one does not have to reinvent the wheel
- Changes to NuTools are closely monitored to ensure that no experiment breaks it for the others
- https://cdcvs.fnal.gov/redmine/projects/nutools/wiki

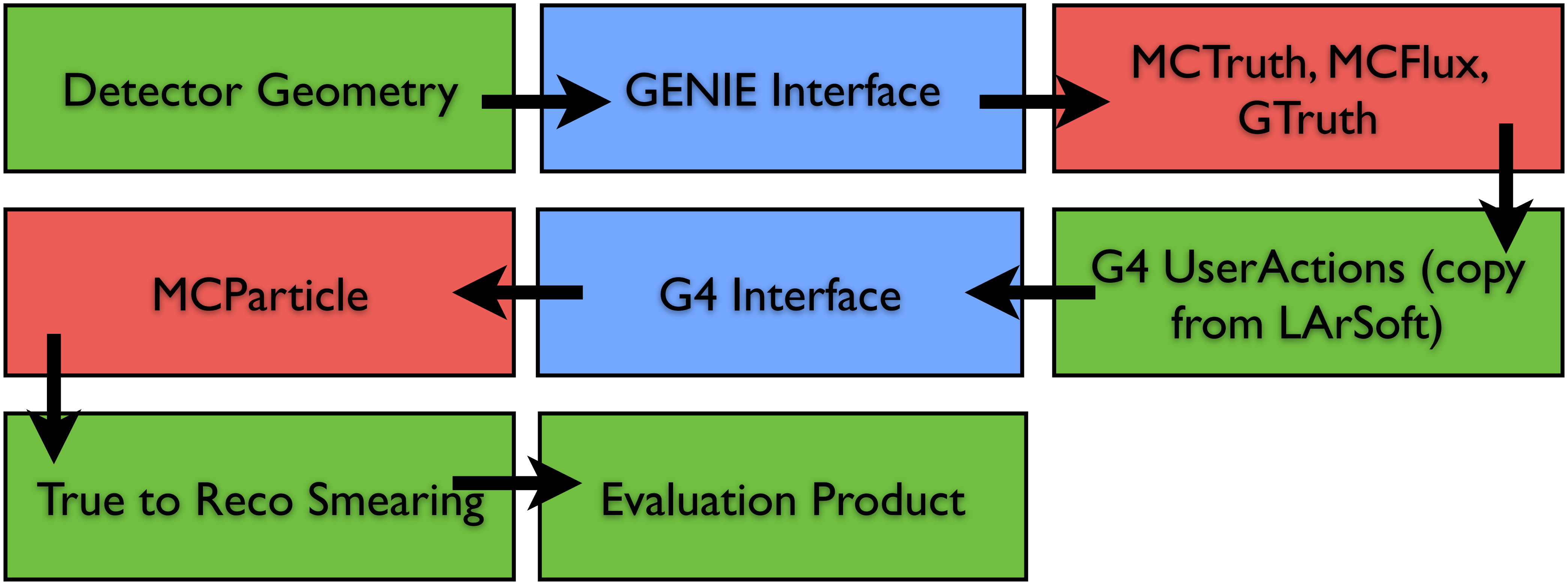
NuTools in Relation to Other Packages



Using NuTools To Evaluate ND Options

- NuTools is already used by the LArTPC folks on DUNE, and folks working on the straw-tube tracker detector are also familiar with it, as are some of the folks interested in the high-pressure gaseous argon TPC option
- This is a set of tools that can be easily deployed and allow for an apples-to-apples comparison of the different options
- The following are the necessary inputs from the groups working on each option
  - A GDML file describing the geometry in a format that is also readable by ROOT (ie some allowed constructs are not currently implemented in ROOT)
  - A set of G4 user-defined "actions" for telling G4 what information to store (ie how to store the particles that are tracked)
  - A way to tell G4 to be sure to track the particles on the appropriate step scale for monolithic volumes (ie LArTPC, HPGTPC)
  - Much of the two areas above can be copied from LArSoft

Work Flow



Supplied by NuTools

NuTools Data Products

Supplied by ND

Getting Started

- The STT and HPGTPC both need to have git repositories setup for storing the code
- Within those repositories some boiler plate definition files need to be created (copied from LArSoft)
  - Some of the files will reside in a ups directory to define the ups product information for the detector-specific code
  - Top level CMakeLists.txt file for the build
  - This strategy keeps the coding environment the same as for LArSoft
- At least two packages need to be created in each repository
  - Geometry to hold the definition of a Geometry art service and the GDML files
  - G4Interface to hold the code needed to communicate to the NuTools G4Base code
  - A separate package to perform the conversion of G4 energy depositions into raw data can be added as appropriate, but not necessary to get started

Making Comparisons

- A module will be run to simulate the energy depositions in each of the detectors
- That module will create data products as defined in the NuTools/Simulation package
- Suggest creating a separate git repository to hold the code that does the comparison between the different detector options
  - Keeps the comparison code consistent between the groups
  - Comparisons could be done using a combination of ART files and TTrees created from ART files

MCTruth

```
namespace simb {

  class MCParticle;

  /// event origin types
  typedef enum _ev_origin{
    kUnknown,           ///< ???
    kBeamNeutrino,      ///< Beam neutrinos
    kCosmicRay,         ///< Cosmic rays
    kSuperNovaNeutrino, ///< Supernova neutrinos
    kSingleParticle     ///< single particles thrown at the detector
  } Origin_t;

  //......................................................................

  /// Event generator information
  class MCTruth {
  public:
    MCTruth();

  private:

    std::vector<simb::MCParticle> fPartList;    ///< list of particles in this event
    simb::MCNeutrino              fMCNeutrino;  ///< reference to neutrino info - null if not a neutrino
    simb::Origin_t                fOrigin;      ///< origin for this event
    bool                          fNeutrinoSet; ///< flag for whether the neutrino information has been set

#ifndef __GCCXML__
  public:

    simb::Origin_t          Origin()           const;
    int                     NParticles()       const;
    const simb::MCParticle& GetParticle(int i) const;
    const simb::MCNeutrino& GetNeutrino()      const;
    bool                    NeutrinoSet()      const;

    void             Add(simb::MCParticle& part);
    void             SetOrigin(simb::Origin_t origin);
    void             SetNeutrino(int CCNC,
                                 int mode,
                                 int interactionType,
                                 int target,
                                 int nucleon,
                                 int quark,
                                 double w,
                                 double x,
                                 double y,
                                 double qsqr);

    friend std::ostream&  operator<< (std::ostream& o, simb::MCTruth const& a);
#endif

  };
}
```

MCParticle

```
namespace simb {

  class MCParticle  {
  public:

    // An indicator for an uninitialized variable (see MCParticle.cxx).
    static const int s_uninitialized; //! Don't write this as ROOT output

    MCParticle();

    // Destructor.
    virtual ~MCParticle();

  protected:
    typedef std::set<int>   daughters_type;

    int                     fstatus;        ///< Status code from generator, geant, etc
    int                     ftrackId;       ///< TrackId
    int                     fpdgCode;       ///< PDG code
    int                     fmother;        ///< Mother
    std::string             fprocess;       ///< Detector-simulation physics process that created the particle
    simb::MCTrajectory      ftrajectory;    ///< particle trajectory (position,momentum)
    double                  fmass;          ///< Mass; from PDG unless overridden Should be in GeV
    TVector3                fpolarization;  ///< Polarization
    daughters_type          fdaughters;     ///< Sorted list of daughters of this particle.
    double                  fWeight;        ///< Assigned weight to this particle for MC tests
    TLorentzVector          fGvtx;          ///< Vertex needed by generater (genie) to rebuild
                                            ///< genie::EventRecord for event reweighting
    int                     frescatter;     ///< rescatter code

#ifndef __GCCXML__
  public:

    // Standard constructor.  If the mass is not supplied in the
    // argument, then the PDG mass is used.
    // status code = 1 means the particle is to be tracked, default it to be tracked
    // mother = -1 means that this particle has no mother
    MCParticle(const int trackId,
               const int pdg,
               const std::string process,
               const int mother  = -1,
               const double mass = s_uninitialized,
               const int status  = 1);


    // our own copy and assignment constructors.
    MCParticle(MCParticle const &)            = default; // Copy constructor.
    MCParticle& operator=( const MCParticle&) = default;

    // Accessors.
    //
    // The track ID number assigned by the Monte Carlo.  This will be
    // unique for each Particle in an event. - 0 for primary particles
    int TrackId() const;
```

MCFlux

```
namespace simb{

  /// Which flux was used to generate this event?
  enum flux_code_{
    kHistPlusFocus  = +1, ///< Flux for positive horn focus
    kHistMinusFocus = -1, ///< Flux for negative horn focus
    kGenerator      =  0, ///< A bogus flux assumed by the generator
    kNtuple         =  2, ///< Full flux simulation ntuple
    kSimple_Flux    =  3  ///< A simplified flux ntuple for quick running
  };

  class MCFlux {

  public:

    MCFlux();

    // maintained variable names from gnumi ntuples
    // see http://www.hep.utexas.edu/~zarko/wwwgnumi/v19/[/v19/output_gnumi.html]

    int    frun;
    int    fevtno;
    double fndxdz;
    double fndydz;
    double fnpz;
    double fnenergy;
    double fndxdznea;
    double fndydznea;
    double fnenergyn;
    double fnwtnear;
    double fndxdzfar;
    double fndydzfar;
    double fnenergyf;
    double fnwtfar;
    int    fnorig;
    int    fndecay;
    int    fntype;
    double fvx;
    double fvy;
    double fvz;
    double fpdpx;
    double fpdpy;
    double fpdpz;
    double fppdxdz;
    double fppdydz;
    double fpppz;
    double fppenergy;
    int    fppmedium;
    int    fptype;     // converted to PDG
    double fppvx;
    double fppvy;
    double fppvz;
    double fmuparpx;
    double fmuparpy;
    double fmuparpz;
    double fmupare;
    double fnecm;
    double fnimpwt;
    double fxpoint;
    double fypoint;
    double fzpoint;
    double ftvx;
    double ftvy;
    double ftvz;
```

MCNeutrino

```
namespace simb {

  //......................................................................

  /// Event generator information
  class MCNeutrino {
  public:

    MCNeutrino();

  private:

    simb::MCParticle fNu;             ///< the incoming neutrino
    simb::MCParticle fLepton;         ///< the outgoing lepton
    int              fMode;           ///< Interaction mode (QE/1-pi/DIS...) see enum list
    int              fInteractionType; ///< More detailed interaction type, see enum list below kNuanceOffset
    int              fCCNC;           ///< CC or NC interaction? see enum list
    int              fTarget;         ///< Nuclear target, as PDG code
    int              fHitNuc;         ///< Hit nucleon (2212 (proton) or 2112 (neutron))
    int              fHitQuark;       ///< For DIS events only, as PDG code
    double           fW;              ///< Hadronic invariant mass, in GeV
    double           fX;              ///< Bjorken x=Q^2/(2M*(E_neutrino-E_lepton)), unitless
    double           fY;              ///< Inelasticity y=1-(E_lepton/E_neutrino), unitless
    double           fQSqr;           ///< Momentum transfer Q^2, in GeV^2

#ifndef __GCCXML__
  public:

    MCNeutrino(simb::MCParticle &nu,
               simb::MCParticle &lep,
               int CCNC,
               int mode,
               int interactionType,
               int target,
               int nucleon,
               int quark,
               double w,
               double x,
               double y,
               double qsqr);

    const  simb::MCParticle& Nu()              const;
    const  simb::MCParticle& Lepton()          const;
    int                      CCNC()            const;
    int                      Mode()            const;
    int                      InteractionType() const;
    int                      Target()          const;
    int                      HitNuc()          const;
    int                      HitQuark()        const;
    double                   W()               const;
    double                   X()               const;
    double                   Y()               const;
    double                   QSqr()            const;
    double                   Pt()              const; ///< transverse momentum of interaction, in GeV/c
```

MCTrajectory

```
amespace simb {

 class MCTrajectory {
 public:
   /// Some type definitions to make life easier, and to help "hide"
   /// the implementation details.  (If you're not familiar with STL,
   /// you can ignore these definitions.)
   typedef std::vector< std::pair<TLorentzVector, TLorentzVector> >  list_type;
   typedef list_type::value_type                     value_type;
   typedef list_type::iterator                       iterator;
   typedef list_type::const_iterator                 const_iterator;
   typedef list_type::reverse_iterator               reverse_iterator;
   typedef list_type::const_reverse_iterator         const_reverse_iterator;
   typedef list_type::size_type                      size_type;
   typedef list_type::difference_type                difference_type;

   /// Standard constructor: Start with initial position and momentum
   /// of the particle.
   MCTrajectory();

 private:
   list_type ftrajectory;

ifndef __GCCXML__
 public:

   MCTrajectory( const TLorentzVector& vertex,
                 const TLorentzVector& momentum );

   /// The accessor methods described above.
   const TLorentzVector& Position( const size_type ) const;
   const TLorentzVector& Momentum( const size_type ) const;
   double  X( const size_type i ) const;
   double  Y( const size_type i ) const;
   double  Z( const size_type i ) const;
   double  T( const size_type i ) const;
   double Px( const size_type i ) const;
   double Py( const size_type i ) const;
   double Pz( const size_type i ) const;
   double  E( const size_type i ) const;

   double TotalLength() const;

   friend std::ostream& operator<< ( std::ostream& output, const MCTrajectory& );

   /// Standard STL methods, to make this class look like an STL map.
   /// Again, if you don't know STL, you can just ignore these
   /// methods.
   iterator               begin();
   const_iterator         begin()      const;
   iterator               end();
   const_iterator         end()        const;
   reverse_iterator       rbegin();
   const_reverse_iterator rbegin()     const;
   reverse_iterator       rend();
   const_reverse_iterator rend()       const;

   size_type size()                    const;
   bool      empty()                   const;
   void      swap(simb::MCTrajectory& other);
   void      clear();
```