

NEUGEN OO Review

Marc Paterno
Mark Fischler
Panagiotis Spentzouris

May 30, 2001

Abstract

This document contains a summary of our findings from the **NEUGEN OO** review, held on 17 April, 2001.

1 Introduction

This document contains a summary of our findings from the review of the **NEUGEN OO** package. This is, in general, not a detailed code review, although some of the reviewers have included detailed analysis of parts of the system. Instead, this document is mostly a high-level overview. It is our aim to provide recommendations for high-level decisions, because of the early state of the software.

The appendix provides some detailed analysis from one of the authors.

2 Purpose of NEUGEN OO

The purpose of **NEUGEN OO**¹ is to serve as a replacement for and extension to the FORTRAN package **NEUGEN**. This FORTRAN package is 15 years old, and has been used by several experiments. It has been developed and maintained by a mid-sized group, and has experienced much growth (in scope of functionality and bulk of code) since it was first created.

Originally written to generate neutrino interactions for studies of backgrounds to proton decay, the existing FORTRAN had new features added as new physics became interesting. Primarily, these new features consisted of new physics processes to be simulated, and extension of the calculations to higher energy scales.

¹A description of the **NEUGEN OO** package may be found on the web, at <http://www.hep.umn.edu/~gallag/neugen>.

While the existing FORTRAN code has clearly been of great use, several reasons for re-writing the package, using an object-oriented design executed in C++, were presented. These reasons were:

- Difficulty of maintenance of the existing FORTRAN code. Over the lifetime of the code, and as a result of modification by many authors, the code has become quite complex and difficult to maintain.
- Difficulty of extension of the code. It is important to be able to add new physics processes without “breaking” existing processes. Looser coupling between different physics processes is required; “dangerous” couplings must be avoided. It is currently very difficult to modify an existing process, or add a new one, without subtly (or perhaps grossly) affecting another process, because of the degree of interaction between the simulations of the existing physics processes.

3 Requirements

In this section, we list physics requirements (the physics scope of the product), functional requirements (what the code must do) and operational requirements (how it must do it). These requirements were all mentioned during the talk; we have tried not to add any of our own prejudices to this list. We follow each section with some brief comments on these requirements.

3.1 Physics Scope

Neutrino physics experiments cover a large range of energies, and thus different regimes of neutrino interactions. For Soudan 2, the energy scale is $O(1)$ GeV; the relevant interactions are ν -nucleus. For NuTeV, the energy scale is $O(100)$ GeV; the relevant interactions are ν -parton. MINOS lies in between; the physics is less well understood, especially in the transition region between the two regimes. There is more than one physics model to choose from for dealing with the transition and low energy regions. This is also the case for the hadronization process. There is uncertainty not just in the model parameters, but in the choice of model. There is uncertainty not just in the parameters, but in the choice of model.

In addition, nuclear effects, even in the high-energy regime, are “layered” upon the free parton interactions — this introduces one of the “dangerous couplings” noted earlier, which reflect on the complexity of the design of the “Summing” algorithms. In addition, there is more than one option (methodology) for the design of such an algorithm.

3.2 Functional Requirements

The main functions for **NEUGEN OO** are:

- Generation of events, representing the interaction of neutrinos with a given “target”.
- Calculation of cross sections for the interaction of neutrinos with a given “target”.

“Target” here implies any object with which neutrinos interact, i.e a parton, a nucleon, a nucleus, an electron, etc.

The system must be easy to use for each of these tasks.

The next most common use of the package will be to test the faithfulness of the simulation. This is more complicated, and will necessarily require more expertise on the part of the user.

Finally, it must be possible to analyze uncertainties in the calculated quantities. This use requires the most sophistication on the part of the user.

For all these tasks, user must have control over the parameters that affect physics output. In addition, the data sets against which the package will be tested have to be available to the users as part of the package, together with concrete examples on their use.

3.2.1 Comments on the Functional Requirements

We heartily endorse the requirement of ease-of-use (and wish the authors of more packages agreed). We suggest that this ease of use should extend not only to running an already-built executable, but should extend to building and configuring the program as well.

We suspect that the task of analyzing uncertainties in the calculated quantities will require considerable thought during the design process. It will be important to determine what sort of uncertainty analysis will be supported, and what degree of user configurability is required. Related issues extend all the way from choosing the specific quantities for which uncertainties shall be calculable, to deciding whether to support “frequentist” uncertainty calculations, “Bayesian” uncertainty calculations, or to allow the user to make his own choice.

Is it necessary to track the user-configurable parameters of the simulation? The answer to this question make all the difference when evaluating solutions for user configuration. Automatic tracking of these parameters is possible, but such a requirement necessarily adds to the complexity of the process of configuring the system. Choosing not to use automatic tracking of parameters adds to the complexity of managing non-trivial generated event samples. While we do not recommend a specific choice, we do recommend that both options be considered.

3.2.2 Documentation of Requirements

We recommend the creation of a requirements document. It is useful to have an up-to-date requirements document, so that at all times a clear reminder of the project priorities is available. The document should be revised as often as necessary to reflect changes in understanding of the project. The intent of such a document is not to limit innovation; it is to support necessary change, by helping to assure the project goals are always clearly defined. This becomes even more important as the number of individuals collaborating on the project grows.

3.3 Operational Requirements

It is important to be able to add a new algorithm in a new run of the program, without recompilation. This flexibility is to allow the user to switch between the many possible physics models for neutrino interactions. A user must be able to easily add his own physics models, without disrupting existing models.

Speed of execution is *not* currently considered a critical feature. This is, in part, because it is expected that any reasonable implementation will be sufficiently fast.

Correctness is, of course, crucial.

It must be easy to interpret the program output, and easy to access underlying physics.

The package must be flexible: it should not be MINOS-specific. It should be able to generate cross-sections, as well as events.

It must be possible to run the generator in “stand-alone” mode. There must be no requirement for a detector simulation, and a detector simulation will not be part of the package.

The physics output of the program must be compared with data to determine the accuracy of the simulation, as well as to determine optimal default settings of parameters.

The software must be well documented. This includes documenting not only the operation of the program, but also documenting the verification of the physics output of the program.

3.3.1 Comments on the Operational Requirements

The importance of flexibility in adding new algorithms suggests that a solution based on run-time polymorphism (an “object-oriented” design), rather than one based on compile-time polymorphism (a “generic programming” design, based on templates) will be most appropriate. This may have an adverse effect on the speed of execution, but with reasonable design effort this should not become a problem. This requirement for flexibility seems to be one of the most important to the

design of the system. It will be important to keep it in mind when evaluating choices for specific design decisions, in each part of the code.

We agree with the philosophy behind the second point. Optimization for speed should be done only after a correctly functioning program exists, and should be guided by measuring (profiling) executables which perform realistic tasks.

Two of the operational requirements are unclear in meaning. To aid in the design process, we suggest that these requirements be specified more clearly; otherwise, it will be impossible to tell if something important is being missed. The following are the requirements we found to be unclear:

1. It must be easy to access underlying physics.
2. The package must not be MINOS-specific.

We recommend they be clarified, in the requirements document mentioned in § 3.2.2.

We recommend that the requirements document include the insistence that associated with any module implementing any physics concept, there be documentation including the equations and/or processes that this module purports to use. We recommend this be in the form of text with equations in properly typeset form (probably this forces it to be snips of \LaTeX) and the requirement must be that the typical MINOS physicist must be able to understand that document to know what the module is doing, and if versed in computing, should be able to check personally that the module is doing what is claimed.

We agree with the author on the importance of quality documentation. We believe that it is important to be specific about the requirements for comparison of physics output with data for each algorithm. It seems to us that this could be extremely difficult, especially for the lower-level algorithms in the system, which may not produce directly visible results. Perhaps it is worthwhile to have several different, but clearly defined, documentation requirements, one for each type of algorithm in the system. These requirements should also be listed in the requirements document.

If you are serious about making testing against data part of the mantra of augmenting this package, then we suggest you can increase the odds of people adhering to that philosophy by providing a standardized form for how such a test must look. A well-thought-out recipe for what constitutes a test, and a uniform structure to the test process, will make the non-physics part of the burden as palatable as possible.

Of course, the original designers must set the right example by always adhering to this mantra themselves! And somebody has to have the right and the gumption to say “this cross section algorithm cannot yet be part of **NEUGEN OO**, until an adequate test against data (or other test) can be devised and is implemented.”

3.3.2 Comments on Extensibility and Flexibility

Much stress was placed on the importance of extensibility and flexibility for **NEUGEN OO**. These are the operational requirements that are most expensive, in terms of design and coding effort. We believe that it is important to understand these requirements clearly, and to keep them in mind, to assure the success of the project — to prevent wasting scarce human resources on less important features, and to prevent missing a crucial feature. To help in understanding the magnitude of the task imposed by these requirements, we have several questions.

In what dimension (or dimensions) is extensibility required? What features can be clearly identified as *not* requiring extensibility? A clear statement of goals on this issue is critical to understanding how much designer effort is required to achieve the goal.

Is the user community supportive of (or does enough of the community even know about) this project? Perhaps it is sensible to collaborate with others who are doing similar work now — or even with people in other experiments, who aren't doing anything now, but who have a (perhaps undiscovered) need for the same product.

What has to be done to make sure that modifications to one part of the system don't destroy some feature of the solution required by another part, or the system as a whole? There are different solutions to how to stitch together solutions to subproblems; these need to be in place, and the class design has to make sure this is handled automatically. To understand this issues, a clear statement of the purpose of each subsystem is necessary. Within each subsystem, a clear statement of the purpose of each class is necessary. Our experience with the review of many systems is that if it is difficult to express the purpose of a given subsystem or class, then the design of that part of the system is probably insufficiently focused, which will lead to problems when extension of the system is required.

The requirements document proposed in § 3.2.2 will be useful in improving the focus of subsystems and classes.

3.4 Weakness of the FORTRAN

In devising the requirements for **NEUGEN OO**, we believe it is important to consider the weaknesses (and strengths) of the previous FORTRAN code. For example, one of the weaknesses pointed out during the review talk was that it was too hard to add new summing algorithms, because each new one needs new types of information. Cross section algorithms are less “changeable”, because what they do is more standardized. What about the list of arguments for cross section calculations, which must specify the initial and final states?

What were the other major failures or problems with the existing FORTRAN? What can the new design do that the old FORTRAN can

not do? Conversely, what things could the old design do that the new design need not support? One item of the latter category mentioned during the review talk (and subsequently questioned) is support for proton decay. The requirements document should address these issues directly; the wisdom accumulated from use of the existing code is very valuable, and should not be lost.

3.5 Use Cases

We believe that, in a small project, the development of formal use case diagrams is of limited utility. We believe a clear requirements document better meets the need for which the use case diagrams were intended.

However, use-case examples serve as both a tool for making certain the design is on-target for how the package is to be used, and later as pedagogical aids for physicists using the package. As such, collection of these examples is a useful part of the design process.

It is clear that most of the design effort (thus far) has gone into the simpler Evaluate Cross Section use case. For example, it seems that the *XSecAlgorithm* interface does not support what would be needed for Generate Event.

3.5.1 Evaluate Cross Section

This is the simpler of the two use cases. The main difficulty we had with this use case is that we are unclear about how one specifies a process for which the cross section is to be evaluated. We suggest that this needs further clarification.

For example, will the cross-section query allow inclusive classifications (neutral current, charged current, elastic, etc), semi-inclusive, or complete determination of the final state (practically impossible, if the hadronization process is considered)? How much flexibility will be allowed in the definition of the kinematic variables of the final state?

3.5.2 Generate Event

This is the more complicated use case, which in turn uses Evaluate Cross Section.

The basic element of Evaluate Cross Section is the *Step*. Each *Step* gets the *Event*, modifies it, and passes it on. This work is done by a set of “algorithms”. Are these the same algorithms as *XSecAlgorithm*? Are they different ones? Do these algorithms need to generate matrix elements? Slide 30 talks about this. It seems that there may be an effort here to unify, under a single interface, things that are unrelated. For example, *target selection* and *hadronization* are clearly distinct operations. Unifying these (as well as others) under a single interface would be useful if some client class is going to use them polymorphically. But

what client will not need to distinguish between an object that does target selection and one that does hadronization? It was unclear to us if we have missed a fundamental feature of the design, or if this is a case of needless abstraction.

It is clear that this use case drives most of the requirements (both functional and operational) of the system. We suggest these requirements be determined in greater detail before significant details of the design are implemented.

3.5.3 Differential cross sections and applying algorithms

You have given a lot of thought to getting total cross sections but now it is necessary to flesh out the processes used in generating decays, in case new design issues arise there. For example, the entire issue of clashes between two interactions, and summation issues, may be different and one should know at an early stage.

4 External Infrastructure

It is important, early in the project, to determine what environments (operating systems and compilers) are to be supported. Clearly it must be the needs of the users that drive these decisions. We caution against the support of compilers that are of pre-Standard vintage, because such compilers generally lack the ability to support modern C++ usage.

It is also important to determine what external products may be relied upon. For example, is it allowable to rely upon any database product (several high-quality free ones are available)²? Upon a commercial database product? ROOT? ZOOM? Other FNAL software, such as RCP?

The requirements document recommended in § 3.2.2 should discuss these requirements and constraints.

5 Suggested Process

In this section, we recommend a process for the continuation of the **NEUGEN OO** project.

First, we reiterate our suggestion (see § 3.2.2) for a requirements document. This document is the most important “design document” for the project.

Second, we suggest that a prototype implementation of the system should be produced at the earliest possible date. It is not important

²The two most widely used examples are probably PostgreSQL and MySQL.

that this prototype support all important features. It is important that it support a few features, at least partially, and that it can serve as a testing-ground for design ideas.

Third, we recommend that design meeting (smaller in scope, and faster in response time, than this review) be used frequently to help keep the project on track (that is, keep it adhering to the requirements document).

If a formal software process is desired, we do not recommend any of the “heavyweight” processes (such as formal UML, or RUP). These processes typically require too much overhead (diagramming, documenting, *etc.*) to be supportable over the long run. Instead, we suggest a model more like Extreme Programming (XP) as being appropriate for a project of this size.³

6 Classes

This section is a collection of notes on the details of several of the classes presented during the review talk. We believe the points made here are valuable, but also recognize that the details of the classes that form the system will be subject to considerable change during the prototyping process.

6.1 Framework

The class *Framework* only deals with “generic” (not the template meta-programming sense) algorithms. Should it know there are different types of algorithms? It is not clear if the level of polymorphism proposed here is useful.

How should the behavior be divided between the *SumAlgorithm* and the *Framework* classes?

How does one make sure that we don’t have a misconfigured? For example, one will want to make sure one is not doing the same “step” twice — except when it makes sense. For which steps does this make sense? If possible, the design should prevent nonsensical configurations at compile time; failing that, at link time; and under all circumstances, at run time.⁴

How often is a new type of interaction added? There are not a huge list, but the ability to add new ones is at least of modest importance. It is unclear if the interactions fall into few enough categories that they may all be known to the system, or if the system must be open to extension to handle unforeseen categories of interactions.

³The requirements document we recommend is really just a collection of *stories* in the XP jargon.

⁴This is consistent with the design of C++ as a strongly-typed language, in which it deemed advantageous to catch errors as early as possible.

It is unclear whether the *Framework* should know about some basic physics, thus helping to make sure that unreasonable things are not done, or if it should know nothing, so that adding new types of interactions is simple. Clarification of the competing requirements is a precondition to making the correct design choice.

6.2 Algorithm

Each *Algorithm* has a name, a version number, and some number bunch of parameters, which have to be sufficient to capture the state of the algorithm represented by the instance of the class. *Algorithm* has nothing algorithm-like about it; it seems to be more of an artifact for persistency.

6.3 XSecAlgorithm

XSecAlgorithms are used to calculate cross sections. What is the interface that supports this calculation? This seems to be the crucial feature of this class, which led to much of the discussion regarding the cross section queries.

It will be important to analyze the requirements for differential cross section calculations, to determine what similarities and differences there are between the interfaces for total and differential cross section algorithms.

Can one *XSecAlgorithm* call another? We suspect this will be relevant both in the design of the interface, and in determining how to handle the interdependency issue raised in § 2.

How are the physics effects factorized? Do “nuclear effects” alter the *XSecAlgorithms*? Are they part of the calculation, or are they added later? How about hadronization? For example, in the case of deep inelastic scattering, is the neutrino-parton interaction a separate calculation from the electroweak radiative corrections, the above two separate from the hadronization process, and all of the above from nuclear effects? Although this model looks like a reasonable way to implement a cross-section algorithm, it breaks in cases like NLO QCD charm production, where the differential cross-section is given by a convolution of fragmentation and DIS. How will differences like that implemented in the design?

6.4 SumAlgorithm

Each *SumAlgorithm* adds up contributions from one or more *XSecAlgorithms*. The *SumAlgorithm* does a great deal of work — it has to understand how to talk to the various *XSecAlgorithms*. This means the interface of *XSecAlgorithm* has to support all the features needed to do

this for every different summing algorithm. Does each new summing algorithm add a new way to have to query the *XSecAlgorithm*? How often do new summing algorithms appear, as compared to new cross section algorithms? Should the design be optimized for ease of addition of new *XSecAlgorithms*, or for adding new *SumAlgorithms*?

Should a *SumAlgorithm* be calling an *XSecAlgorithm*, or should it just get passed results from the *XSecAlgorithm*?

7 Miscellaneous Questions and Comments

This section contains a collection of otherwise unsorted questions or comments. Many are items which arose in discussions during the review meeting, and are recorded here for completeness, and in case they are significant for further discussion.

1. Do the “initial state” and “final state” objects represent a range of inputs? So far, this has not been needed, because the design only deals with total cross sections, rather than differential cross sections. The differential cross section problem is not yet solved.
2. Should **NEUGEN OO** use Slunits? This seems like an excellent choice, unless it is precluded by a need to support older compilers, or some other technical reason prevents its use.
3. How does IO work? What gets to be persistent, and how? What requirements can be placed on user code to support persistence?
4. Is one event always within one medium? What sort of media should we be able to deal with? How do the media get into the system? What geometric ideas does the system have to understand? Can we make it need to understand no such things? What processes are not point-like, and thus require geometric information?

A Concepts Related to interaction types, overlaps, clashes

This section has been written by Mark Fischler.

As I see it, here is a problem:

The various routines which apply an interaction and give a contribution to the cross-section are not independent of one another. The approach suggested by the code given at the review has the major drawback that higher-level classes need to know about all the types of interactions (as evidenced by the symptom of that enum and case statement). This in itself makes addition of a new interaction type tricky. Worse,

if the summation routines start needing to deal with overlaps between different types of interactions, then changes will start to require looking at $O(N^2)$ places because of N^2 potential overlaps.

The goal in resolving this would be to:

Allow for creation of additional interaction types by creation and instantiation of new classes derived from a base *InteractionType* class. This should *not* involve any alterations of higher classes.

Define properties regarding these interface classes, such as whether a clash of two classes of the same type in the same context is permissible.

For the cases where it is necessary to define what to do when two types of interactions are present define some object encapsulating that. Code would instantiate an object derived from that *Overlap* base class to provide the method to apply when this happens.

When a solution to this has been designed and coded, before it is used extensively, I strongly suggest you let someone like Paterno, Kowalkowski, or in a pinch myself review just that one narrow issue.

To do this, I propose a structure something like the following:

There is a class *InteractionType* — each type of interaction, for example *DeepInelasticScattering*, inherits from *InteractionType*. A very simple registry mechanism is set up to provide for each class derived from *InteractionType* to have a unique id (which for these examples I will assume should be an int). I give a sample of how this could be done below; superior experts may suggest improvements on this scheme but it works.

It turns out that *InteractionType* is exactly at the level in the hierarchy where you have *XSecAlgorithm*, so from now on, I will assume that it is named *XSecAlgorithm*. Specific *XSecAlgorithms*, which are particular cases of an interaction type, then inherit from *XSecAlgorithms*, for example:

```
class DIS : public XSecAlgorithm { /* ... */ };
class mySpecificDIS : public DIS { /* ... */ };
class alternativeDIS : public DIS { /* ... */ };
class QEL : public XSecAlgorithm { /* ... */ };
class mySpecificQEL : public QEL { /* ... */ };
// ...
```

XSecAlgorithm also contains these virtual methods:

```
class XSecAlgorithm {
// ...
virtual bool isOverlapAllowed {return false;}
virtual XSec overlap (const XSecQuery & q,
                    const XSec & xA,
```

```

                                const XSec & xB);
};

```

and of course the methods you already have:

```

class XSecAlgorithm {
// ...
virtual XSec getXSec (XSecQuery q);
// ... etc.
};

```

Normally, overlaps of two of the same type of interaction are verboten: The main framework can query each existing interaction type class for id, and if two match, it should call `isOverlapAllowed()` which if nothing else is done will return false. The bool method `isOverlapAllowed()` can be overridden in a derived class to allow overlaps.

So how does usage look? Well, declaring a new *XSecAlgorithm* was fine before and works just about the same way, except that now it inherits from the appropriate *XSecAlgorithm* instead of setting its int-*Type* data member. It still the same has implementations of all the real physics methods.

Creating a whole new *XSecAlgorithm*, which previously was awkward in that the main framework had to change to know about the new type, is now straightforward. You follow this trivial boilerplate:

In the header:

```

class XYZ : public XSecAlgorithm (
    static int id;
public:
    int myId() const {return id;}
    // ... plus all the methods you normally need
}

```

and in the source file:

```

int XYZ::id = XSecAlgorithmRegistry::id();

```

I suggest that *XSecAlgorithm* ought to contain, besides the virtual bool `isOverlappedAllowed()`, a virtual method

```

virtual XSec overlap (const XSecQuery & q,
                    const XSec & xA,
                    const XSec & xB      ) const;

```

This would be called if two interactions of the same type are defined, and if `isOverlapAllowed()` is overridden to return true for each of these. What this does is it gives you a way to resolve that situation

by inspecting the XSecs returned by each and forming from them the actual XSec to use. (I would augment XSec to include some sort of confidence or weight number to help this overlap routine decide who to pay the most attention to.)

This takes care of (in the class design sense of avoiding changes propagating to affect classes they shouldn't; not in the physics thought sense!) the case of multiple XSec algorithms of the same type. It is also plausible to extend the technique to take care of cases where type ABC and type XYZ affect one another, again without having to change the main code each time such a cross-effect is added. Thus potentially your summation routines can look a lot cleaner.

OK, here is how the code can look. This compiles and correctly does what is needed: The `myId()` method returns the id assigned to the class XYZ, which is unique with regard to all other classes derived from *XSecAlgorithm*. Again, I don't claim this can't be improved, but it proves that you can avoid the major drawback that higher-level classes need to know about each type of interaction.

A.1 Headers

```
// XSecAlgorithmRegistry.h
class XSecAlgorithmRegistry {
public:
    static int id() {return max++;}
private:
    static int max;
};

// XSecAlgorithm.h
class XSecAlgorithm {
public:
    XSecAlgorithm();           // As per your code
    // more methods           ... As per your code
    virtual int myId() const {return id;}
    virtual bool isOverlapAllowed() const
        {return false;}
    virtual XSec overlap (const XSecQuery & q,
                        const XSec & xA,
                        const XSec & xB) const ;

private:
    static int id;
}

// DIS.h
class DIS : public XSecAlgorithm {
public:
```

```

XSecAlgorithm();          // As per your code
// more methods          ... As per your code
virtual int myId() const {return id;}
// And ***optionally***
virtual bool isOverlapAllowed() const {return true;}
virtual XSec overlap (const XSecQuery & q,
                     const XSec & xA,
                     const XSec & xB) const ;

private:
    static int id;
}

// QEL.h
class QEL : public XSecAlgorithm {
public:
    XSecAlgorithm();          // As per your code
// more methods          ... As per your code
virtual int myId() const {return id;}
// Note that isOverlapAllowed() and overlap() can be
// omitted if not relevant
private:
    static int id;
}

```

A.2 Source Files

```

// XSecAlgorithm.cc
// Your methods already present, plus...
int XSecAlgorrithm::id = XSecAlgorithmRegistry::id();
XSec XSecAlgorithm::overlap (const XSecQuery & q,
                             const XSec & xA,
                             const XSec & xB) const {
// default algorithm, if overlap is permitted, yet no
// specific overlap method was provided, is very simple,
// like...
if ( xZ.weight() > xB.weight() ) {
    return xA;
} else {
    return xB;
}
}

// DIS.cc
// Your methods already present, plus...
int DIS::id = XSecAlgorithmRegistry::id();
virtual XSec overlap(const XSecQuery & q,

```

```
        const XSec & xA,  
        const XSec & xB) const {  
    XSec answer;  
    // insert real overlap physics here to compute answer!  
    return answer;  
}  
  
// QEL.cc  
// Your methods already present, plus...  
int QEL::id = XSecAlgorithmRegistry::id();
```