Application of Unit Tests for Genie

James Jones

SIST Intern, Summer 2016

Fermilab National Accelerator Laboratory
Department of Scientific Computing
Supervisor: Gabriel Perdue
August 18, 2016

ABSTRACT

Test-driven development is a type of software development in which the user creates certain tests that the software needs to be able pass before the software is designed. This assures that the software is developed the way it needs to be and allows the developer to debug while they are designing the software. One such test driven development strategy is the unit test. A series of tests designed to test pieces of the designed software piece by piece

My goal this summer was to look into the application of Unit Tests for GENIE, the Neutrino Monte Carlo Generator, which has no type of testing for it. The following contains a brief intro to GENIE, as well as the research and application process that went into designing such Unit Tests.

INTODUCTION TO GENIE

GENIE is a suite of products (*Generator*, *Comparisons*, *Tuning*) for the experimental neutrino community. GENIE uses a framework neutrino Monte Carlos (a series of computational algorithms) in order to simulate detailed experimental setups involving neutrinos, from implementing flux drivers to calculating cross sections from the collision between a neutrino and a target. This allows users to create their own hypothetical experiments and run them on GENIE in order to see what happens. It can also be used by physicists already working on a neutrino experiment in order to both predict the data that will be gathered as well as make sure that the data gathered matches what is supposed to happen. The GENIE project is added onto an updated by a global group of physicists working on the major neutrino experiments. Presently, GENIE is being used by a majority of neutrino experiments, including those using the JPARC and NuMI neutrino beam lines. GENIE takes on an important role in the design and execution of these experiments. It is used to evaluate the feasibility of proposed projects and estimate their physics impact, make decisions about detector design and optimization, analyze the collected data samples, and evaluate systematic errors. It is projected to be an important physics tool for the exploitation of the world accelerator neutrino program [1] [2].

The GENIE software uses a collection of classes that contains certain data and algorithms that can be accessed and used, depending on what the user requests done via the command line interface. For example, the cross section model in GENIE provides the calculation of differential and total cross sections. During the event generation the total cross section is used together with the flux to determine the energies of interacting neutrinos. The cross sections for specific processes are then used to determine which interaction type occurs, and the differential distributions for that interaction model are used to determine the event kinematics. [1].

```
60 double LwlynSmithQELCCPXSec::XSec(
61     const Interaction * interaction, KinePhaseSpace_t kps) const
62 {
63   if(! this -> ValidProcess    (interaction) ) return 0.;
64   if(! this -> ValidKinematics (interaction) ) return 0.;
65
66   // Get kinematics & init-state parameters
67   const Kinematics &   kinematics = interaction -> Kine();
68   const InitialState & init_state = interaction -> InitState();
69   const Target & target = init_state.Tgt();
70
71   double E  = init_state.ProbeE(kRfHitNucRest);
72   double E2 = TMath::Power(E,2);
73   double ml = interaction->FSPrimLepton()->Mass();
74   double M  = target.HitNucMass();
75   double q2 = kinematics.q2();
76
77   // One of the xsec terms changes sign for antineutrinos
78   bool is_neutrino = pdg::IsNeutrino(init_state.ProbePdg());
79   int sign = (is_neutrino) ? -1 : 1;
80
```

Figure 1: Piece of code to a cross section calculation shows the inter-play between classes.

GENIE is a publically available program and can be operated via the command line interface. For example the code for generating a cross section is:

gmkspl –p # -t # -o myxsec.xml:

-p specifying the neutrino PDG code, -t specifying the target PDG code, and –o

specifying the name of the XML file that these cross sections can me written in to. For example the instruction to calculate the Xsec splines for a muon neutrino (PDG code: 14) scattered of Fe^{56} (PDG code: 1000260560) and place it in an XML file called cross_sections.xml is:

gmkspl –p 14 –t 1000260560 –o cross_sections.xml

Once these instructions are entered, GENIE is able to use its interconnected classes to gather the data it needs and produce the correct cross sections for such a hypothetical interaction.

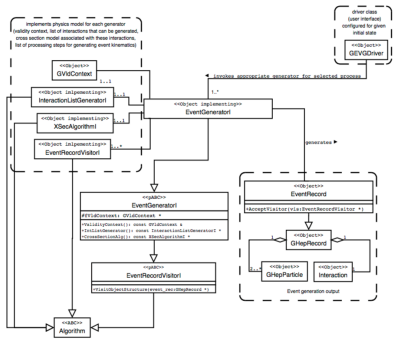


Figure 2: GENIE's individual classes contain unique information and functions that can be accessed depending on the task.

INTORDUCTION TO UNIT TESTING

Unit testing is a method for software testing in which fragments of code are tested to determine if they are fit for use. In this case unit usually refers to the smallest piece of code such as a variable definition or a return value. Unit testing can be used to test things such as classes, variables or functions in order to determine that they work correctly and return what is expected to be returned [4]. Such tests are applied throughout the code in order to make sure that every fraction of the code is working appropriately. For example, part of a unit test within GENIE could test the XSec() function within the LlewellynSmith class to make sure that the function returns the correct cross section value.

While Unit testing can be very useful, it comes with both benefits and certain limitations. The benefits lie in the fact that applying a unit testing method to ones code, will allow errors and mistakes to be caught earlier and easily. It also allows one to be able to modify their code without fear, for if created properly, a unit test should be able to tell if the modified code works as well as the original. However, unit testing is limited in the fact that a unit test will only be as thorough as the programmer designs it to be. A very thorough unit-testing program calls for many hours of tedious, and boring work; hours that a programmer may not want to put in. Even if the unit test is thorough, there is still a probability that mistakes may be able to get past the tests, if it's not thorough enough and the wrong things are tested. This would call for the programmer to return and redesign their tests.

Each Unit test is built using a unit test framework; functions and languages that allow for easier unit testing. In order to do this for GENIE, which uses C++, a language that does not have its own built in framework, I had to research, choose and download an already existing framework that could be combined with and work coherently with C++.

BOOST

BOOST is a set of libraries for C++ that gives C++ programs the ability to support things such as, number

generation, linear algebra and unit testing. After doing research on C++ unit testing frameworks, I decided to use the BOOST unit-testing framework because it had the most advantages over disadvantages from the recommended frameworks. Within the BOOST unit-testing framework it is easy to add new tests and to test things within the code. It is also able to handle crashes well. The one disadvantage that comes with BOOST is that it requires the programmer to include some external libraries before creating the unit test.

```
BOOST_AUTO_TEST_CASE( my_test )
{
    // seven ways to detect and report the same error:
    BOOST_CHECK( add( 2,2 ) == 4 );        // #1 continues on error
```

Figure 3: BOOST has it's own functions which make creating unit tests easy.

Within the BOOST unit-testing framework there are BOOST functions, which are designed specifically for running tests within the unit test and for separating and organizing tests. Before discussing the functions it is important to discuss the organization tools:
BOOST_AUTO_TEST_SUITE
BOOST_AUTO_TEST_CASE.
These are used to organize the unit test so that like functions can be tested. For example, if I wanted to test all physics related functions within my code I would create a BOOST_AUTO_TEST_SUITE named physics. Then if within this suite I wanted to functions having to do with acceleration and functions having to do with force, I would create two BOOST_AUTO_TEST_CASEs named, acceleration and force. It is within these test cases that the functions can be tested for correctness.

```
BOOST_AUTO_TEST_SUITE(Physics)
BOOST_AUTO_TEST_CASE(Acceleration)
{
        // Here we would test acceleration based funcions.
}
BOOST_AUTO_TEST_SUITE(Force)
{
        // here we  would test functions having to do with force.
}
BOOST_AUTO_TEST_SUITE_END
```

Figure 4: Shows how BOOST can organize test types for more efficient testing.

Utilizing both the test suites and the test cases are useful for organizing the unit-test and will allow users to more easily determine where errors are and what they might be.

Once the suites and cases are set up it is within the cases that the programmer can test certain aspects of their code. In order to do this, BOOST has created functions of their own that check different things and respond in different ways. These functions are used to check the validity of the functions given to them.

```
int add( int i, int j ) { return i+j; }

BOOST_AUTO_TEST_CASE( my_test )
{
    // seven ways to detect and report the same error:
    BOOST_CHECK( add( 2,2 ) == 4 );        // #1 continues on error

    BOOST_REQUIRE( add( 2,2 ) == 4 );      // #2 throws on error

    if( add( 2,2 ) != 4 )
      BOOST_ERROR( "Ouch..." );            // #3 continues on error

    if( add( 2,2 ) != 4 )
      BOOST_FAIL( "Ouch..." );             // #4 throws on error

    if( add( 2,2 ) != 4 ) throw "Ouch..."; // #5 throws on error

    BOOST_CHECK_MESSAGE( add( 2,2 ) == 4,  // #6 continues on error
                         "add(..) result: " << add( 2,2 ) );

    BOOST_CHECK_EQUAL( add( 2,2 ), 4 );   // #7 continues on error
}
```

Figure 5: Examples of the BOOST unit-test functions

In Figure 5 several different BOOST unit-test functions can be seen, each having different requirements and responses. The simplest of these functions is BOOST_CHECK. BOOST_CHECK can be used to make sure a function has the correct output, such as BOOST_CHECK (add(2,2) == 4), which

makes sure that the function that adds two numbers works correctly. It can also be used with Booleans to make sure that they return true: BOOST_CHECK (physics.isvalid()).
When the BOOST_CHECK function encounters an error it displays a default error message, including the source file name, source file line umber, and expression that failed. BOOST_CHECK differs from the other functions in that if the function encounters an error it will report it and move on instead of stopping the unit-test[3].

BOOST_REQUIRE is a function that works exactly like BOOST_CHECK. However if BOOST_REQUIRE encounters and error it does not move on, but instead reports the error and terminates the program. This function is likely to be used over BOOST_CHECK when testing a function that is imperative to the rest of the code and who's failure would make future testing unfeasible or inaccurate.

BOOST_ERROR is a function that does not detect an error, but instead can work with another function such as an IF statement to produce a specialized error message to the user. This can be used if the programmer has a specific message they want to send about the function being tested. Once the BOOST_ERROR is used, like BOOST_CHECK, the unit-test continues to run. However, BOOST_FAIL is another function like BOOST_ERROR, accept when it is used the unit-test is halted [3].

```
if( add( 2,2 ) != 4 )
  BOOST_ERROR( "Ouch..." );

if( add( 2,2 ) != 4 )
  BOOST_FAIL( "Ouch..." );
```

Figure 6: An example of BOOST_ERROR and BOOST_FAIL being used.

The last two simple BOOST unit-test functions, work like a combination of the already existing ones. BOOST_CHECK_MESSAGE works like a combination of BOOST_CHECK and BOOST_ERROR, checking a function to see if it works and then generating a custom message after it does so. BOOST_CHECK_EQUAL works like BOOST_CHECK but is made specifically to test if the two arguments given to it are equal. While all of these BOOST unit-test functions are alike, their minor differences allor programmers to be able to customize how they want their unit-test to work and what they want it to look, like.

APPLYING BOOST UNIT TESTING

After building BOOST and learning how to use its unit-testing framework, my next step was to figure out how to combine it with GENIE so that we can apply such a unit-testing framework. Creating a unit test for GENIE is not difficult. The difficult part is working with all of GENIE's classes in order to build the correct classes and functions so that the right values can be tested.

Our first attempt to get a proper working unit-test with GENIE involved me trying to test a cross section value (xsec) in the function XSEC of the LlewellynSmith class. In the file LwlynSmithQELCCPXSec.h the function Xsec can be found as well as the arguments it needs.

```
double XSec           (const Interaction * i, KinePhaseSpace_t k) const;
double Integral       (const Interaction * i) const;
bool   ValidProcess   (const Interaction * i) const;
```

Figure 7: The XSec function with its arguments.

To begin the program we first included the BOOST library that grants us access to the unit test framework, all the header files of the classes that we knew the function XSEC relied on.

```
#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE Genie
#include <boost/test/unit_test.hpp>
#include <iostream>
#include <iterator>
#include "LlewellynSmith/LwlynSmithQELCCPXSec.h"
#include "Interaction/Interaction.h"
#include "Interaction/KPhaseSpace.h"
#include "Interaction/XclsTag.h"
#include "Interaction/Kinematics.h"
#include "PDG/PDGUtils.h"
```

Figure 8: BOOST unit-test frameworks and class header files are included.

Before working with the XSec function (figure 7), it is important to realize the arguments that it takes and figure out how to provide those before calling the function. It can be seen that XSec takes pointer to a GENIE class called "interaction" and another variable, KinePhaseSpace_t k, which (through some serious digging) we discovered is actually just a number. So before calling the XSec function we had to create a pointer to the class and create a variable that is the same number as KinePhaseSpace_t k.

```
BOOST_AUTO_TEST_SUITE(Lwlyn)
    // Build things from class Interaction for testing
    LwlynSmithQELCCPXSec * lwlyn = new LwlynSmithQELCCPXSec();
    Interaction * inter = new Interaction();
    KinePhaseSpace_t kps = kPSQ2fE;
    // int A nd Z represent carbon
    int A = 12;
    int Z = 6;
    int probe_pdgc = 14;
    int recoil_nuc;
    bool isP;
    int target_pdgc = pdg::IonPdgCode(A,Z);
```

Figure 9: The pointer to the interaction class and the KinePhaseSpace variable are initiated with other variables when the BOOST suite first opens

Once the variables and pointers are set up, assuming they are done correctly, the desired number should be able to be accessed and tested by the unit-test.

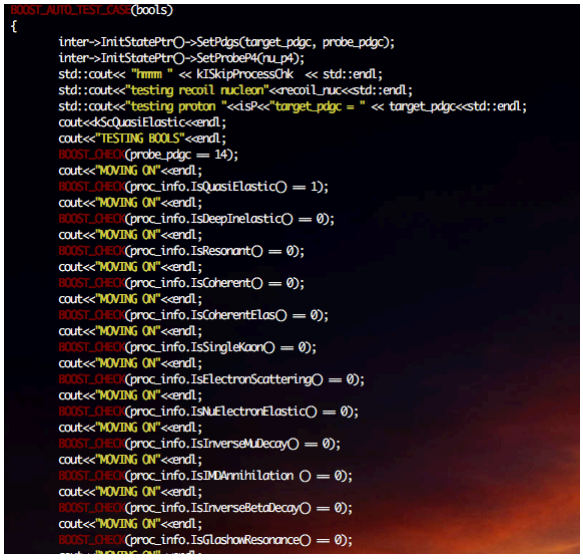
```
BOOST_AUTO_TEST_CASE(doubles)
{
inter->InitStatePtr()->SetPdgs(target_pdgc, probe_pdgc);
inter->InitStatePtr()->SetProbeP4(nu_p4);
std::cout << "The XSec is " << lwlyn->XSec(inter, kps) << std::endl;
BOOST_CHECK(lwlyn->XSec(inter, kps) == 5.3365984441833672E-11);

}
```

Figure 10: The pointers are set up and the function is tested in BOOST

By calling on the XSec function in the LlewellynSmith class and placing that inside the BOOST_CHECK function, one can check that whatever that function returns is what it should be.

Theoretically, if all the classes and variables are instantiated correctly, this should be how to test a function in the LlewellynSmith class. However, when attempted the XSec function reported that the cross section was 0, which means that something in the function failed because a piece of the code was not initiated correctly. Using gdb, it was discovered that the XSec function failed because a function within another class failed. ValidProcess is a boolean within the LlewellynSmith class. At first we believed

that it failed because we hadn't included the correct classes; but after a long period of trial and error (almost three weeks) we still could not figure out why Valid Process failed. In the end we decided to try a simpler test, in order to prove that a BOOST unit-test works with GENIE. This new test was designed to test only Booleans in the ProcessInfo class, using BOOST_CHECK to simply see if the Booleans returned a 1 or a 0. This test was began the same way the previous one had been, including the same classes and initiating the same pointers. However, because we were only testing Booleans there was nothing that could fail so the test was successful, proving that it is possible to design a unit-test for GENIE using the BOOST framework.



```
BOOST_AUTO_TEST_CASE(bools)
{
        inter->InitStatePtr()->SetPdgs(target_pdgc, probe_pdgc);
        inter->InitStatePtr()->SetProbeP4(nu_p4);
        std::cout<< "hmmm " << kISkipProcessChk << std::endl;
        std::cout<<"testing recoil nucleon"<<recoil_nuc<<std::endl;
        std::cout<<"testing proton "<<isP<<"target_pdgc = " << target_pdgc<<std::endl;
        cout<<kScQuasiElastic<<endl;
        cout<<"TESTING BOOLS"<<endl;
        BOOST_CHECK(probe_pdgc == 14);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsQuasiElastic() == 1);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsDeepInelastic() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsResonant() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsCoherent() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsCoherentElas() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsSingleKaon() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsElectronScattering() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsNuElectronElastic() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsInverseMuDecay() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsIMDAnnihilation () == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsInverseBetaDecay() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsGlashowResonance() == 0);
```

Figure 11: The redesigned unit-test tested only Booleans in the ProcessInfo class

Upon proper compilation and running of the program is designed to tell the user if any errors were encountered and where they were encountered. This allows the user to easily find and fix whatever is wrong with the program. For example, figure 12 shows the results of running the code in figure 10. It can be seen that when checked to see if they returned false, two of the Booleans actually return true, which means either something is wrong with the Booleans or the unit-test should have checked if they returned true instead of false.

```
TESTING BOOLS
MOVING ON
MOVING ON
MOVING ON
MOVING ON
MOVING ON
MOVING ON
MOVING ON
MOVING ON
MOVING ON
MOVING ON
MOVING ON
MOVING ON
MOVING ON
MOVING ON
MOVING ON
MOVING ON
MOVING ON
gtest1.cpp(81): error in "bools": check proc_info.IsWeak() == 0 failed
MOVING ON
gtest1.cpp(83): error in "bools": check proc_info.IsWeakCC() == 0 failed
MOVING ON
MOVING ON
CLEAR
TESTING target_pdgc
CLEAR
```

Figure 12: The results of running the unit-test code from figure 10

While this is only a small unit-test, testing only a tiny fragment of the GENIE code, it shows that using a BOOST unit test for GENIE is possible and how it would work.

FUTURE WORK AND PROBLEM SOLVING

After confirming that BOOST unit-tests can be used for GENIE, the next step would be to create an infrastructure so that people who use GENIE can also use the unit-test program along with it. This involves using base class with inheritable functions that setup the needed classes and run the unit-test. This is what I am currently working on as of August 10, 2016 with about a week and a half left of the program.

However, there are still some problems and mysteries with BOOST that

need to be solved before any designed unit-test can run perfectly. The first is that the BOOST unit-test framework does not run with int main(). In fact, if int main() is included anywhere within the program, the test will not run. This makes it a bit difficult to try and do other things at the same time that the unit-test is running. However, the solution to this could simply be to research more into the BOOST syntax.

Another problem encountered while working with the unit-test was getting the GENIE unit-test to work without including the entire GENIE code itself. This required using pointers to open up the right classes for use and initiating the right variables, so that the function that needed to be tested could run properly. This isn't impossible, but is tricky and takes time, due to the fact that one mistake can mess up the program, as well as the fact that some functions are very deep; they involve several different classes that need functions from other classes that need functions from other classes and so on. This is the reason Valid Process, in the xsec test, failed; We did not go deep enough in our investigation of the multitude of functions it relies on in order to work properly. Therefore it is the job of the programmer to "go down the rabbit hole", and investigate and initiate this chain of classes until they reach the bottom. This is not impossible, but very time consuming.

CONCLUSION

It can be seen BOOST unit-tests can work for GENIE and be quite simple. Although it would take a good amount of time, it is possible to create a BOOST unit-test that tests all of GENIE. Once developed and applied, the GENIE unit-test program will allow scientists to quickly identify errors in their code, as well as maybe change some pieces of the program and test that to see if it works (that depends on the level of detail of the unit-test program). While it is not necessary, test-based development is a beneficial part of the scientific computing community, for those who are willing to put in the time and effort to make sure such tests work.

References

[1] Audreopoulos et al. "The GENIE Neutrino Monte Carlo Generator: Physics & User's Manual" http://projects-docdb.fnal.gov/cgi-bin/RetrieveFile?docid=753&filename=GENIE_PhysicsAndUserManual_20100927.pdf&version=5

[2] "GENIE Neutrino Monte Carlo" http://genie.hepforge.org

[3] "Unit Test Framework: User's Guide" http://www.boost.org/doc/libs/1_56_0/libs/test/doc/html/utf/user-guide.html

[4] *Xie, Tao*. *"Towards a Framework for Differential Unit Testing of Object-Oriented Programs"* (PDF)