MONTE CARLO EVENT GENERATION IN A MULTILANGUAGE, MULTIPLATFORM ENVIRONMENT

N. A. Graf, A. Johnson, SLAC, Menlo Park, CA 94025, USA

Abstract

We discuss techniques used to access legacy event generators from modern simulation environments. Coding to a standard interface and use of shared object libraries enables runtime selection of generators, and allows for extension of the suite of available generators without having to rewrite core code.

PROBLEM STATEMENT

Physics and detector simulations for the International Linear Collider are being conducted by an amorphous and heterogeneous group of high energy physicists, working mostly part-time on this project. Simulation software needs to be lightweight, yet flexible and performant over a wide variety of development platforms. The HEP community has mostly completed its transition to modern, object-oriented programming languages such as C++ and Java, e.g. GEANT4, ROOT and JAS. One exception is event generators; most event generators producing unweighted events with stable, final state particles appropriate for detector response simulations are either written in FORTRAN or depend on FORTRAN-based libraries for fragmentation, e.g PYTHIA, HERWIG, ISAJET, or pandora-pythia and whizard [1]. Interfacing to these legacy programs presents an additional level of complexity to software development and distribution.

INTERFACING LEGACY EVENT GENERATORS

Most generators can target the HEPEVT FORTRAN common block. The stdhep [2] package provides a binary persistence binding, but stdhep is not well supported on all platforms, and somewhat tricky to implement for casual users. One solution would be to simply interface to the event generators through their persistent output, with a standalone FORTRAN program producing and writing events to disk using the stdhep file format. The end user application would then just read this file and process its contents. However, disk storage can be prohibitive in many large-statistics fast-simulation physics analyses. End users would also be directly exposed to the idiosyncrasies of each package, i.e. no standard main programs or runtime input commands exist. The alternative presented here is to abstract out a common event generation interface and provide standard implementations for some of the more popular event generation packages.

A Pure Java Solution

Providing a simulation and reconstruction framework written in Java has proven to be well matched to the ILC developers environment. It is Object Oriented and easy to use, with many support libraries and is platform independent. However, no full event generators are written in Java. We do, however have a pure Java diagnostic event generator for generating simple events composed of final state particles. This has proven to be very helpful in determining the intrinsic detector response (e.g. energy, momentum and position resolution, and geometrical acceptances) for single particles.

A Multilanguage Solution

The solution presented here uses Java to provide a consistent platform-independent interface. We use the Java Native Interface (JNI) [3] to call FORTRAN event generators via C++. The use of shared-object (.so) or dynamic link (.dll) libraries enables late binding and provides runtime flexibility.

Encapsulation

The philosophy adopted in this approach is to encapsulate the basic event generation behavior and to push as much responsibility as possible onto the native event generators (note that this is different from the approach taken in [4]). Runtime control is achieved through plain-text ASCII files which are parsed by FORTRAN routines just as they would be in an all-FORTRAN environment. These communicate with the COMMON blocks in the event generator code to set parameters, reusing each package's existing parsing code. The end user can select physics processes, beamstrahlung, decay channels, etc. at runtime simply by editing a text file. The user can either generate events on the fly within a fast-simulation and analysis environment, or write out events in stdhep format. The Java program calls a single C++ class through JNI with a minimal interface, viz.

```
public native void initialize();

public native void nextEvent(
        int[] nev,
        int[] isthep, int[] idhep,
        int[] jmohep, int[] jdahep,
        double[] phep, double[] vhep);

public native void finish();
```

The C++ code communicates with FORTRAN only through the HEPEVT common block, which is mapped on to the hepevtcommon struct as follows.

```
extern "C" void initialize_();
extern "C" void nextevent_();
extern "C" void finish_();

typedef struct // HEPEVT common block
{
// event number
 int nevhep;
// number of entries
 int nhep;
// status code
 int isthep[4000];
// PDG particle id
 int idhep[4000];
// position of 1st, 2nd mother
 int jmohep[4000][2];
// position of 1st, last daughter
 int jdahep[4000][2];
// 4-momentum, mass
 double phep[4000][5];
// vertex position and time
 double vhep[4000][4];
} hepevtcommon;
```

The FORTRAN event generation code simply fills the HEPEVT common block for each event. This code, of course, has to be written by someone with expertise in the generator, but then can be used by anyone. Control is through the *initialize()* call, and is generator-specific.

Example Program

```
import hep.analysis.EventGenerator;
import hep.analysis.EventData;
import hep.analysis.EndOfDataException;
import hep.io.stdhep.adapter.StdhepAdapter;

/**
* LCD interface to stdhep event generators.
* @author Norman Graf
*/

public class StdhepEventGenerator extends EventGenerator
{
   StdhepAdapter _stdadapter;
   private EvtGen _eventgen;

   /**
   * Constructor loads native library
   */
   public StdhepEventGenerator(String generatorName)
   {
      _stdadapter = new StdhepAdapter();
      System.loadLibrary(generatorName+"evtgen");
      _eventgen = new EvtGen();
      // Note that the constructor calls initialize()
   }

   /**
   * Generate a single event
   */
   public EventData generateEvent() throws EndOfDataException
   {
      _eventgen.nextEvent();
      return _stdadapter.convert(_eventgen.genStdhepEvent());
   }

   /**
   * Finish up
   */
   public void afterLastEvent()
   {
      _eventgen.finish();
   }
}
```

Runtime Control

This solution interacts natively with the event generators, thereby reusing existing parsing code where available. For instance, ISAJET has a well-defined set of control cards which completely specify the event generation characteristics and a well-established mechanism for reading these at runtime, which we have respected. The input file is exactly the same as for the FORTRAN job, allowing users to reuse their existing command files. PYTHIA has a command-parsing capability which allows many of the runtime switches to be set by simply passing string commands to the PYGIVE subroutine. HERWIG has neither a well-defined set of runtime controls, nor a native mechanism for dynamically setting event characteristics, so in this case we abstracted out a reasonable set of parameters and allow the user to modify these as needed. Other event generators are handled similarly, and clearly needs input from an expert in the generator. The interaction between Java and C++ is only through the *initialize()* method. All .dll or .so libraries respect the same interface, so the user can dynamically select and load packages at runtime, even those which were not available at link time, viz.

```
>java EvtGen libToLoad
```

The catalog of available generators can be expanded in the future, without users having to modify any of their existing code. Simply distribute a new .so or .dll file and its corresponding run control file.

SUMMARY

The use of Java as the user interface and JNI to connect to legacy FORTRAN event generation code has proven to be a useful solution to the problem of accessing legacy event generation code written in FORTRAN from simulation software written in Java and run on multiple platforms. The definition of a minimal interface allows event generators to be selected at runtime without code modification and enables smooth future upgrades.

REFERENCES

[1] http://www.thep.lu.se/ torbjorn/Pythia.html
http://hepwww.rl.ac.uk/theory/seymour/herwig
http://www.phy.bnl.gov/ isajet
http://www.slac.stanford.edu/ mpeskin/LC/pandora.html
http://www-ttp.physik.uni-karlsruhe.de/whizard

[2] http://www-cpd.fnal.gov/psm/stdhep/

[3] http://java.sun.com/j2se/1.5.0/docs/guide/jni/

[4] http://www.slac.stanford.edu/econf/C0303241/proc/papers/THJT005.PDF