LU–TP 00–23
hep-ph/0006152
May 2000

Pythia version 7-0.0 – a proof-of-concept version*

Marc Bertini, Leif Lönnblad and Torbjörn Sjöstrand
Department of Theoretical Physics
Lund University
Sölvegatan 14A
S-223 62 Lund, Sweden
marc@thep.lu.se, leif@thep.lu.se, torbjorn@thep.lu.se

Abstract

This document describes the first proof-of-concept version of the Pythia7 program.

Pythia7 is a complete re-write of the Pythia program in C++. It is mainly intended to be a replacement for the 'Lund' family of event generators, but is also a toolkit with a structure suitable for implementing any event generator model.

In this document, the structure of the program is presented both from the user and the developer point of view. It is not intended to be a complete manual, but together with the documentation provided in the distribution, it should be sufficient to start working with the program.

*Work supported by the EU Fourth Framework Programme 'Training and Mobility of Researchers', Network 'Quantum Chromodynamics and the Deep Structure of Elementary Particles', contract FMRX-CT98-0194 (DG 12 - MIHT).

Contents

1 Introduction **3**
- 1.1 Motivation . . . 3
- 1.2 Strategy . . . 4
- 1.3 About this document . . . 4
- 1.4 Future plans . . . 5

2 Description **5**
- 2.1 Installation . . . 5
- 2.2 Running the sample generator . . . 6

3 User information **6**
- 3.1 The event record . . . 6
- 3.2 Particle properties . . . 7
- 3.3 Setting up an event generator run . . . 8
  - 3.3.1 The handler classes . . . 8
  - 3.3.2 Sub-process selection . . . 10
  - 3.3.3 Constructing the event . . . 11
  - 3.3.4 The line-mode user interface . . . 11
- 3.4 Running an event generator . . . 12
  - 3.4.1 The standard run program . . . 13
  - 3.4.2 Using an event generator in another application . . . 13

4 Developer information **13**
- 4.1 The class structure . . . 13
  - 4.1.1 Reference counted classes . . . 13
  - 4.1.2 Persistent classes . . . 14
  - 4.1.3 Interfaced classes . . . 15
  - 4.1.4 Other classes . . . 16
- 4.2 The repository . . . 16
  - 4.2.1 Interfaces . . . 16
  - 4.2.2 Parameters . . . 17
  - 4.2.3 Switches . . . 17
  - 4.2.4 Commands . . . 18
  - 4.2.5 References between objects . . . 19
  - 4.2.6 Isolating an event generator . . . 19
- 4.3 Creating new handler classes . . . 21
  - 4.3.1 Partonic cross-sections . . . 22
  - 4.3.2 PDF's and remnant handling . . . 25
  - 4.3.3 Luminosity functions . . . 26
  - 4.3.4 Step handlers . . . 27
  - 4.3.5 Cascade handlers . . . 29
  - 4.3.6 Hadronization handlers . . . 29

4.3.7 Decay handlers . . . 29
4.3.8 The Standard Model . . . 29
4.3.9 Random Numbers . . . 30
4.3.10 Exceptions . . . 30
4.4 Utility classes . . . 31
4.4.1 Lorentz5Vector . . . 31
4.4.2 Math functions . . . 32
4.4.3 Selector . . . 32
4.4.4 SimplePhaseSpace . . . 32
4.4.5 Other utility classes . . . 32
4.5 Documentation . . . 33

5 Case study : The Lund String Fragmentation 33
5.1 The Lund fragmentation algorithm . . . 34
5.2 The design analysis . . . 34
5.3 The `LundFragHandler` class . . . 35
5.4 Outlook and comparison with PYTHIA version 6 . . . 36

6 Bugs, lacking features and future plans 37

1 Introduction

PYTHIA7 [1] will be a new event generator well suited to meet the needs of future high-energy physics projects, for phenomenological and experimental studies. The main target is the LHC community, but it will work equally well for linear e^+e^- colliders, muon colliders, the upgraded Tevatron, and so on. The generator will be based on the existing Lund program family, but rewritten from scratch in a modern, object-oriented style, using C++. The greatly enhanced structure will make for improved ease of use, extendibility to new physics aspects, and compatibility with other software likely to be used in the future.

1.1 Motivation

The current state-of-the-art event generator programs from the Lund group — PYTHIA, JETSET and ARIADNE — generally work well. It has been possible gradually to extend them well beyond what could originally have been foreseen, and thus to parallel the development of the high-energy physics field as a whole towards ever more complex analyses. However, a limit is now being approached, where a radical revision is necessary, both of the underlying structure and of the user interface.

Even more importantly, there is a change of programming paradigm, towards object-oriented methodology. In the past, particle physicists have used Fortran, but now C++ is taking over. C++ has been adopted by CERN as the main language for the LHC era. The CERN program library is partly going to be replaced by commercial products, partly be rewritten in C++. In particular, the rewriting of the detector simulation program Geant [2] is a major ongoing project, involving several full-time programmers for many years, plus voluntary efforts from a larger community. The CERN shift is matched by corresponding decisions elsewhere in the world: at SLAC for the B-factory, at Fermilab for the Tevatron Run 2, and so on.

Therefore a completely new version of the Lund programs, written in C++, is urgently called for. What is required is a complete rethinking of the way an event generator should look.

Some attempts have already been made. Although the MC++ program[3] never left the toy stage, pieces of it still live on, since a subset of the classes became the foundation of CLHEP[4]. Lately there has been some programs written for parts of the event generation process, e.g. APACIC/AMEGIC[5], and a couple of attempts to implement a general event record in C++[6, 7], but PYTHIA7 is the first serious attempt to write a complete event generator in C++.

1.2 Strategy

The main idea is to define a structure which encapsulates the event generation process. The standard way of generating events is first to choose a hard scattering sub-process from parton densities and hard parton-parton matrix elements. The partons in the scattering are then allowed to develop perturbative showers, and the resulting partons are hadronized. Finally the produced hadrons are decayed until only stable particles remain. There is normally a natural ordering of these steps given by the physical scale involved in the corresponding processes, but the order in which different steps are performed is not always straightforward, and the event generation process can become very complicated e.g. when multiple interactions are included.

In PYTHIA7, all the different steps are described in terms of a set of abstract base classes with well defined interfaces using virtual member functions. Any model for a specific step in the event generating process can then be implemented in a class which inherits from the appropriate base class. In addition, any such model can be modified by further inheritance. There are several such base classes in PYTHIA7, some of which are shown below in fig. 2.

A number of models implemented like this can then at run-time be combined into an `EventGenerator` which can produce `Event`s. An `Event` is, of course, also an object representing the produced particles. But the `Event` class is not just a list of particle entries, as the *event records* of current Fortran-based generators. It is a highly structured class containing the complete history of the generation of each event. Of course, if the user is only interested in a list of final state particles, this is also easily extracted.

1.3 About this document

The layout of this paper is as follows. Section 2 describes how to obtain, install and run a sample PYTHIA7 program. Section 3 contains a more detailed description of how to use the program, while section 4 describes how to implement models in the PYTHIA7 framework. In section 5, the implementation of Lund string fragmentation is given as a case study. Finally in section 6 there is a description of things presently lacking in the program and of bugs which have not yet been corrected.

This is not intended to be a complete manual, but in the HTML version of this paper, which is included in the program distribution, all classes mentioned are linked to html-ized header files, and together they should provide a reasonable guide for anyone who wants to use the program and/or implement physical models.

1.4 Future plans

It should be noted that the program described here is not of production quality and results produced with it should not be used for serious scientific studies. This release is mainly intended to make public the basic structure of the program.

The work to implement real physical models into this framework has started. String fragmentation is e.g. already present (see section 5 and ref. [8]). The plan is to have something of production quality within a year from now, although it may take several years before it can fully make obsolete the current Fortran version of PYTHIA.

2 Description

To run the current PYTHIA7 program, a computer running Linux and the GNU GCC compiler version 2.95.2 is needed. The aim is to have PYTHIA7 written fully according to the ANSI/ISO C++ standard, so in principle it should be possible to run it on any computer with a standard-compliant compiler. However, at present there are very few (if any) fully standard compliant compilers available and, as a consequence, inadvertently, there may be some non-standard code in PYTHIA7. To avoid problems, it is therefore recommended to use PYTHIA7 with the GCC compiler.

To run PYTHIA7 you also need to have the CLHEP library installed, which is available from `http://wwwinfo.cern.ch/asd/lhc++/index.html`.

2.1 Installation

The program is available on the web at
`http://www.thep.lu.se/Pythia7/Pythia7-0.0.tar.gz`.
Unpack the distribution on a computer running Linux by doing
`gunzip -c pythia7-0.0.tar.gz | tar xf -`
This will produce a directory called `Pythia7-0.0/Pythia7`.

To compile, you first need to run the `configure` script in this directory. This will set up the *Makefiles* to use `g++` as compiler and to look for the CLHEP include and library files in the `include` and `lib` directories under `/usr/local`. These choices can be changed by setting the shell environment variables `CXX` and `CLHEPPATH` to the desired C++ compiler and to the path where CLHEP is installed respectively. This should be done before the configure script is run. After this, `make current` will compile the main PYTHIA7 library file and a couple of sample programs in the `src` subdirectory.

2.2 Running the sample generator

There are three sample programs in the `src` subdirectory:

- `setupPythia SimpleLEP.in` reads in the default *repository* of objects and the commands in `SimpleLEP.in` to setup a simple `EventGenerator` object capable of generating $e^+e^- \to q\bar{q}$ annihilation events at 91 GeV. This generator is then written to a file called `SimpleLEP.run`.
- `runPythia SimpleLEP.run` reads the generator and runs it. This will produce three output files: `SimpleLEP.log` will contain a printout of the first ten events and possible error messages; `SimpleLEP.out` will contain the results of the run – in this case only the integrated cross section; and `SimpleLEP.tex` will contain a description of the models used in the run including the relevant references.
- Finally, there is the `runPartial.cc` program which is an example of how to use PYTHIA7 in other applications where full events are not required. `runPartial SimpleLEP.run` will simply hadronize a simple partonic system specified in the program.

The `setupPythia` and `runPythia` programs will be run automatically with `make check`.

3 User information

The normal way to use PYTHIA7 is to run a setup program to define an `EventGenerator`. This generator can then be used in an application to generate events to be used there. But it is also possible to setup a `FullEventGenerator`, so that the full run, including analysis, can be defined. In this case the `runPythia` program can be used to do everything.

3.1 The event record

To inspect and analyze a generated event the user is given an `Event` object which contains a very detailed account of how the event was generated. The structure of the event classes is shown in fig. 1. At high luminosity hadron colliders an event usually consists of several collisions and consequently the `Event` object has a list of `Collision` objects. Each `Collision` has a list of `SubProcess` objects, one primary `SubProcess` and, in case of multiple interactions, a number of secondary ones. The `Collision` has also a list of `Step` objects, each of which has a list of `Particle`s corresponding to the full state of the event after a given step in the event generation process. The last one of these represents the final-state particles which may be detected.

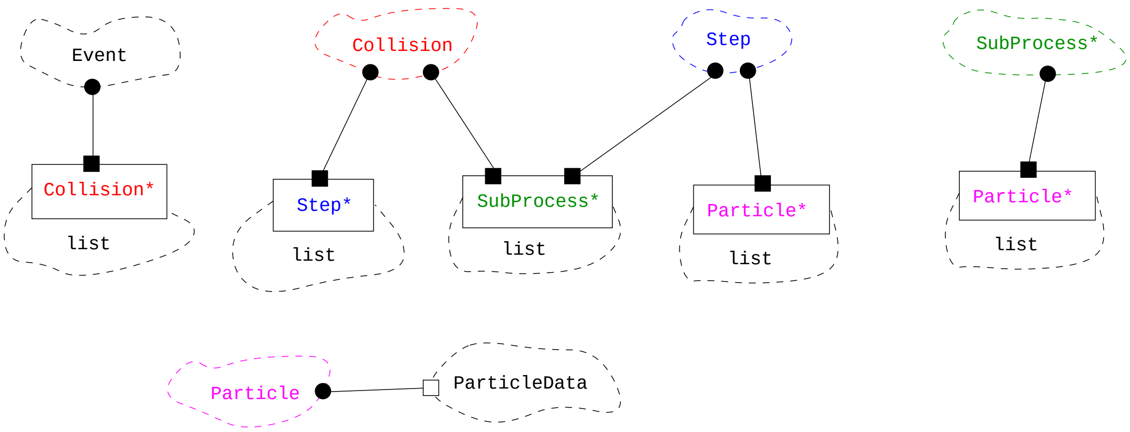

Figure 1: *Class diagram for the structure of a generated event in* PYTHIA7*.*

This is a fairly complicated structure, to allow for detailed analysis of the generation. But the normal user need not worry about the inner structure of an `Event`. Instead there are a number of member functions which can be used to extract the information needed. The main such method is called `select`. Given a so-called predicate object, it will extract particles which satisfy the conditions specified in the predicate into a container. The only requirement on the predicate object is that it is derived from the `SelectorBase` class. PYTHIA7 will contain a number of such selector classes, the most common one being the `SelectFinalState` which simply extracts a list of all final state particles.

The `Particle` class is fairly straightforward. It contains information about an instance of a particle, such as its momentum and creation point, while all properties of the corresponding particle type can be accessed through a pointer to a `ParticleData` object.

The `Particle` object also contains a list of (pointers to) its parents and, in case it has decayed, a list of children. In case the particle is coloured, it also contains pointers to its (anti-) colour neighbours and pointers to the parents and children from/to which it has obtained/given its (anti-) colour.

In some cases the state of a particle instance may have been changed in a generation step, even if it has not decayed, e.g. its energy may have been modified in order to conserve the total energy in some process. For this reason, a particle may have a pointer to objects corresponding to the previous and/or next instance of the same particle.

3.2 Particle properties

The `ParticleData` objects contain all information about a particle type: its id-number (according to the PDG standard[9]), its name (for which there is no standard yet), its nominal mass and width, its charge, spin and colour quantum numbers, etc.

`ParticleData` also has a list of `DecayModes`, each of which, in the simplest form, contains a branching fraction, a list of `ParticleData` objects corresponding to the decay products and a pointer to a `Decayer` object which is capable of performing the decay. But a `DecayMode` may also represent more abstract decay modes such as $\mathrm{W}^+ \to$ *hadrons* or $\mathrm{B}^+ \to \mu^+\nu_\mu +$ *anything*, and also decay chains such as $\mathrm{H}^0 \to \mathrm{W}^+\mathrm{W}^- \to \mathrm{e}^+\nu_\mathrm{e}\bar{\mathrm{u}}\mathrm{d}$. This is achieved by the introduction of *particle matcher* objects (of base class `MatcherBase`), which represent a whole set of particles, and by specifying `DecayModes` of the decay products.

The `ParticleData` object may also have a pointer to a `MassGenerator` object which can generate the mass of a given particle instance, given the nominal mass and width. Also a width generator[2] may be assigned to a `ParticleData` object to calculate the width and partial widths of a particle according to some (e.g. SUSY) model.

A particle may have different kinds of masses. The mass given in the `ParticleData` class corresponds to the kinematical mass, i.e. $\sqrt{E^2 - p^2}$, of a real particle. Quarks and di-quarks may be given as objects of the `ConstituentParticleData` class which inherits from `ParticleData` and carries information about the *constituent* mass. Currently there is no mechanism for specifying other kinds of masses for particles, e.g. scale-dependent masses, but this will be possible in the future.

3.3 Setting up an event generator run

To setup an event generator run, one has to manipulate the `Repository` which has a list of all objects available for use in PYTHIA7. The idea is then to connect different objects with each other (e.g. assign a `MassGenerator` to a `ParticleData` object above). Through the `Repository` it is also possible to modify parameters and switches which may be available in the objects (e.g. the nominal mass in a `ParticleData` object).

3.3.1 The handler classes

The `Repository` will by default contain a large number of sample `EventGenerators`, so the ordinary user should never have to setup an `EventGenerator` from scratch. In fig. 2 is shown the structure of the main handler classes in PYTHIA7. As seen, there are two `EventGenerator` sub-classes, the `FullEventGenerator` and the `PartialEventGenerator`. The latter is a scaled down version which is not able to generate full collisions but can be used inside other applications to apply `StepHandlers` to a user-supplied initial `Step`.

[2] Not yet implemented.

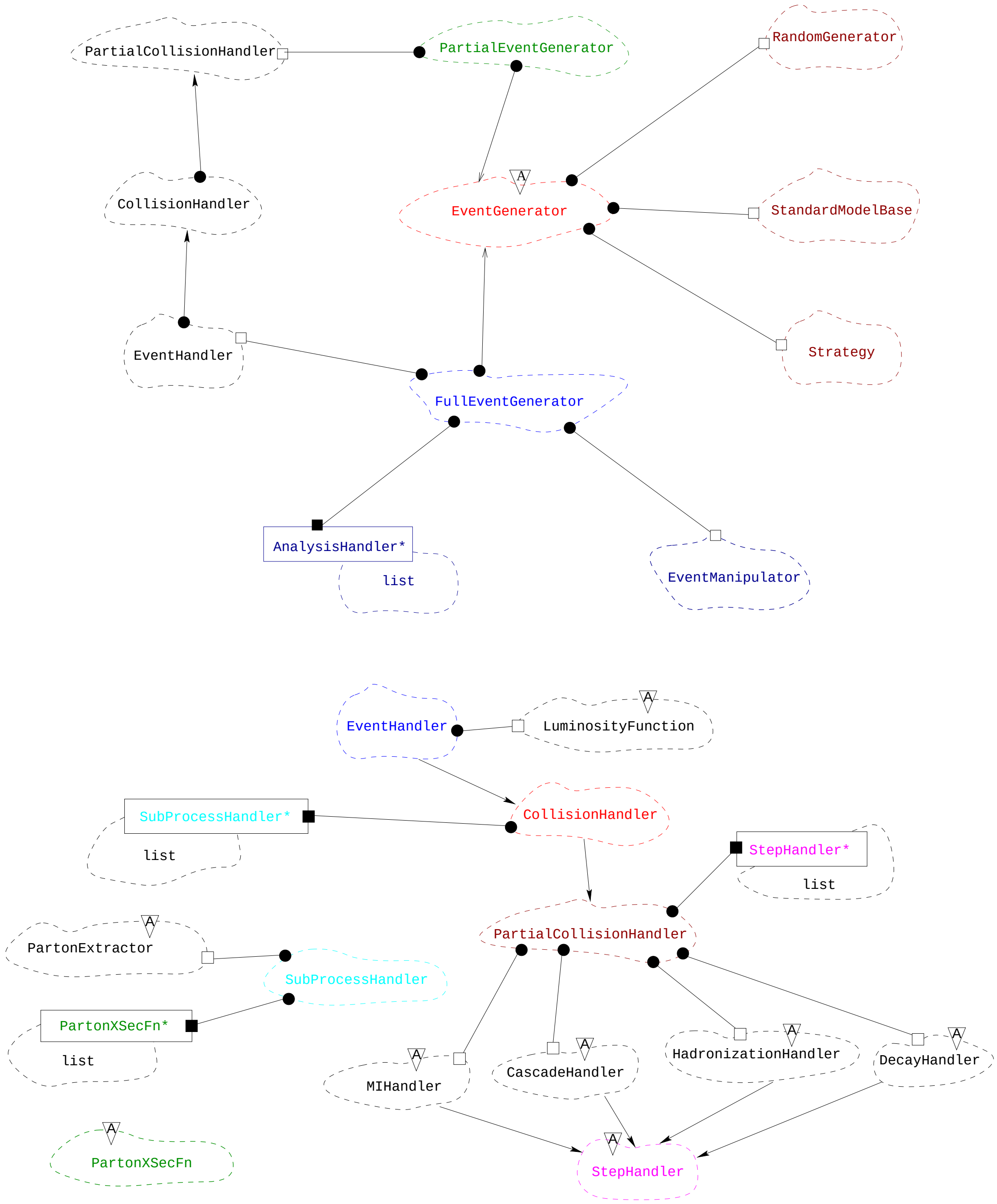

Figure 2: *Class diagram for the structure of different handlers responsible for the generation process in* Pythia7.

Most importantly, the `FullEventGenerator` has a pointer to an `EventHandler` which performs the actual generation of an event. In addition the `EventGenerator` contains objects which are global to a particular run, such as a `StandardModelBase` object which implements the Standard Model parameters to be used, a `Random-Generator` object, the list of `ParticleData` objects to be used, etc. Some of these global objects may be collected in a `Strategy` object.

3.3.2 Sub-process selection

The `EventHandler` uses a `LuminosityFunction` to obtain a pair of colliding beam-particles which are then used by the inherited `CollisionHandler` to perform the actual collision. The `CollisionHandler` may have several `SubProcessHandlers` to manage different kinds of sub-processes, each of which has a `PartonExtractor` object and a list of `PartonXSecFn` objects. The `PartonExtractor` handles the extraction of partons from the beam-particles and the generation of remnants, while the `PartonXSecFn` implements hard parton–parton scatterings. The `Parton-Extractor` can be assigned parton density objects of class `PDFBase`, but normally the colliding particles are given as `BeamParticleData` objects, in which case a `PDFBase` object is specified there. To each `PDFBase` there is assigned a `Remnant-Handler` object which is used by the `PartonExtractor` to generate the remnants.

The selection of a hard sub-collision is a two-step procedure. The `EventHandler` has a list of so-called `XCombs`, one for each possible combination of `SubProcess-Handler`, `PartonXSecFn` and pair of incoming partons. Given a `KinematicalCuts` object, an `XComb` is chosen according to an approximate upper limit of the integrated cross-section for the corresponding sub-process (calculated in the initialization)

$$\sigma^{\max} = \int \frac{dx_1}{x_1}\frac{dx_2}{x_2} x f_{1,i}^{\max}(x_1) x f_{2,j}^{\max}(x_2) \hat{\sigma}_{ij\to X}^{\max}(\hat{s} = S_{\max} x_1 x_2), \tag{1}$$

where x_1 is the momentum fraction of parton i, $xf_{1i}^{\max}$ is an approximate upper limit of the corresponding momentum density (and similarly for 2 and j), $\hat{\sigma}_{ij\to X}^{\max}$ is an approximate upper limit of the hard $ij \to X$ sub-process and $S_{\max}$ is the maximum invariant mass squared of the colliding particles. The $xf^{\max}$ functions are typically normal parton density functions, but may also be convoluted with momentum distributions of the incoming particles, if these do not have a fixed momentum. This convolution may be in several steps, if e.g. the partons come from a resolved photon which comes from an electron with varying momentum.

When an `XComb` has been selected, a phase-space point (x_1, x_2) is generated using the approximate parton densities and $\hat{\sigma}^{\max}$. The chosen `XComb` is then to be accepted with a probability given by the ratio of the exact functions to the approximate upper limits in the phase-space point selected:

$$w = \frac{xf_{1,i}(x_1)}{xf_{1,i}^{\max}(x_1)} \frac{xf_{2,j}(x_2)}{xf_{2,j}^{\max}(x_2)} \frac{\hat{\sigma}_{ij\to X}(S_{\max}x_1x_2)}{\hat{\sigma}_{ij\to X}^{\max}(S_{\max}x_1x_2)}. \tag{2}$$

To calculate this ratio, the `PartonXSecFn` and `PartonExtractor` objects may generate internal degrees of freedom. The Monte Carlo approximation of the integrated cross section for each `XComb` can then be obtained by

$$\sigma \approx \sigma_{\mathrm{MC}} = \frac{N_{\mathrm{accepted}}}{N_{\mathrm{attempted}}} \sigma^{\mathrm{max}}. \tag{3}$$

This approximation will become better the more events are generated.

At present there is no possibility to generate weighted events in PYTHIA7, but such facilities will be provided in the future.

3.3.3 Constructing the event

After an `XComb` has been accepted, the rest of the generation is assumed to continue with unit probability, although any of the subsequent steps in the generation may veto the selected phase-space point.

First the `PartonXSecFn` and `PartonExtractor` are asked to generate their internal degrees of freedom, and the first `Step` of the event is constructed. This may be vetoed according to user-specified `KinematicalCuts` which can be assigned to the `CollisionHandler`.

The `CollisionHandler` inherits from `PartialCollisionHandler`, which implements the list of all `StepHandlers` which should be applied after the initial `Step` has been set up by the `SubProcessHandler`. These handlers are divided into three main groups handling parton cascades, hadronization and particle decays, respectively[3]. Each of these groups has one special-purpose handler and two lists of `StepHandlers` to be executed before and after the main one. The special-purpose handlers, `CascadeHandler`, `HadronizationHandler` and `DecayHandler`, are all derived from the general `StepHandler` class.

3.3.4 The line-mode user interface

The `Repository` is supposed to be accessed by a user interface to handle the setup. But the `Repository` itself also implements a rudimentary line-mode interface which is used in the sample `setupPythia` program. The list of objects in the `Repository` are structured like a Unix file system with directories and subdirectories of objects. The `Repository` communicates with these objects via so-called interfaces which may represent a parameter, a switch or a reference to another object. The *address* of an interface is given as

```
/dir/subdir/.../object:interface
```

As an example, consider an object of the class `ParticleData` called `Z0`, residing in a directory called `/Particles/`. It will have a member variable corresponding to the nominal mass, for which an interface is defined called `NominalMass`. This

[3]These three may, in the future, be expanded with a fourth group for multiple interactions.

parameter can then be accessed through
`/Particles/Z0:NominalMass`

The commands available to manipulate objects are listed in the `Doc/Commands.html` file included in the program distribution. Here is a small subset of them:

`ls dir`
: print the names of the objects in the specified directory.

`describe object`
: print a brief description of the specified object, listing the names of the interfaces defined for it.

`describe object:interface`
: print the description for an interface of an object.

`set object:interface value`
: set an interface of an object to a given value.

`setdef object:interface`
: set an interface of an object to its default value.

`send object:command-interface message`
: send a message to a command-type (see section 4.2.4) interface of an object.

`get object:interface`
: get the value of an interface of an object.

`def object:interface`
: get the default value of an interface of an object.

`min object:interface`
: get the minimum allowed value of an interface of an object.

`max object:interface`
: get the maximum allowed value of an interface of an object.

`saverun run-name eventgenerator-object`
: isolate an `EventGenerator` object together with all objects it refers to, directly or indirectly, and save them to a file called `run-name.run`. Obviously, the changes in the repository by the commands above are propagated to this file.

3.4 Running an event generator

When an `EventGenerator` has been set up and saved to a file it can be used in a number of ways.

3.4.1 The standard run program

The `runPythia` program simply reads a `FullEventGenerator` from a file and calls it's `go()` method, which will run the number of events specified in the generator. For each event a number of `AnalysisHandler`s will be called, again specified in the generator. In addition an `EventManipulator` object may be specified e.g. to handle the running of several different models of some parts of the generation on the same basic collision.

3.4.2 Using an event generator in another application

The same generator object can be read into any other application, where each call to the `shoot()` method will generate an `Event` to be used by the application. In addition, the generator can be used to only perform a subset of the `StepHandler`s on a user-supplied initial `Step` using the `partialEvent()` method, as exemplified in the `runPartial.cc` program.

4 Developer information

In this section, some of the classes are described in somewhat more detail to explain how one goes about implementing new models in PYTHIA7. Further information needed can be found in the header files of the respective classes.

4.1 The class structure

The most important classes are shown in fig. 2. These classes are all *interfaced*, *persistent* and *reference counted*. These are the base categories of classes special to PYTHIA7, which are used besides the standard library classes, classes from CLHEP, and small utility classes.

4.1.1 Reference counted classes

Most classes in PYTHIA7 are reference counted, inheriting from `ReferenceCounted`, to avoid memory leaks in cases it is not obvious who owns an object and is responsible for its deletion. By using a smart pointer class `RCPtr`[4] together with reference counted objects, the user need never worry about having to delete an object when it is no longer needed - this is done automatically when there are no more smart pointers refering to it. This means that the syntax for creating an object is a bit modified:

[4] defined in the namespace `Pythia7::Pointer`, which is normally imported to the `Pythia7` namespace.

```
RCPtr<AClass> p = new_ptr(AClass());
```

A reference counted object may still be created the normal way

```
AClass * p = new AClass;
```

but then someone has to be responsible for deleting the object.

Also an object which is reference counted can be pointed to by a normal pointer:

```
RCPtr<AClass> p = new_ptr(AClass());
AClass * p2 = p;
```

but, of course, no guarantees are given that such a pointer will always point to an existing object. Normal pointers could e.g. be used as function arguments to avoid the overhead of incrementing and decrementing the reference count, but it is recommended that the class `TransientRCPtr<AClass>` is used instead. This is a trivial wrapper around a bare pointer, but can be easily assigned to/from `RCPtr<AClass>`.

There are also classes called `ConstRCPtr<>` and `TransientConstRCPtr<>` for pointers to `const` object. Throughout PYTHIA7 the pointers used are `typedef`'ed using a template class defined as

```
template <typename T>
struct Ptr {
  typedef RCPtr<T> pointer;
  typedef ConstRCPtr<T> const_pointer;
  typedef TransientRCPtr<T> transient_pointer;
  typedef TransientConstRCPtr<T> transient_const_pointer;
};
```

i.e. `Ptr<AClass>::pointer` is used instead of `RCPtr<AClass>`. This is to facilitate a transition to other smart pointer strategies.

4.1.2 Persistent classes

One of the serious shortcomings of the C++ language is that there is no standard way of writing objects to disk in a *persistent* way so that they can be read in again. This is especially troublesome for a program such as PYTHIA7, where the objects are highly interconnected. For this reason, PYTHIA7 includes its own I/O system based on *persistent* streams.

A `PersistentOStream` object is able to write objects to a file in a way such that it can be read in again by a `PersistentIStream`. The persistent streams can read and write basic types just as the standard C++ I/O streams. But in addition they also know how to read and write (smart) pointers to objects derived from the `PersistentBase` class. They are also able to read and write simple classes consisting of basic types and/or pointers to `PersistentBase` objects if the corresponding `<<` and `>>` operators are defined.

To make a class persistent it is not enough to inherit from `PersistentBase`. One must also specify a `ClassDescription` object, specialize the templated `ClassTraits` class and implement special read and write methods. The first two items are handled semi-automatically with the help of macros. The read and write methods must be implemented by hand. This is, however, quite straightforward: Consider a class, `AClass`, which inherits directly or indirectly from `PersistentBase`. If this class has data members `m1`, `m2` and `m3`, which are either basic types, pointers to persistent objects or simple classes containing either, the following non-virtual public methods need to be defined:

```
void AClass::persistentOutput(PersistentOStream & os) const {
  os << m1 << m2 << m3;
}
```

and

```
void AClass::persistentInput(PersistentIStream & is, int version) {
  is >> m1 >> m2 >> m3;
}
```

where the integer argument in the latter can be used to check that the version of the class reading in the object is the same as the one writing it out. Note that no separating white-spaces are needed, as for the standard C++ iostreams.

The persistent streams rely on the ability to construct an object of a class given only its name, and also to access the inheritance relationships between classes. But since the C++ standard does not specify how this should be done – it does not even specify a platform-independent way of getting the name of a class – a number of other things need to be done for classes which are to be persistent, as described below in section 4.3.

The `PersistentOStream` and `PersistentIStream` are developed from the `HepPOStream` and `HepPIStream` classes described in ref. [4].

4.1.3 Interfaced classes

Classes which are to be handled by the `Repository` should inherit from the `Interfaced` class, which in turn inherits from `PersistentBase`. Besides the persistent read and write methods, a concrete interfaced class must implement a `clone` method returning a full copy of the object, straightforwardly implemented as:

```
Ptr<InterfacedBase>::pointer AClass::clone() {
  return new_ptr(*this);
}
```

Optionally the class may implement protected methods called `doinit` and `dofinish`. The `doinit` method is called just before an object is used in a generator run and should be implemented if any initialization is needed. If the initialization fails, an `InitException` should be thrown. `dofinish` is called for all objects used in a generator run after all events have been generated and can e.g. be used to calculate and write out statistics or, in case of `AnalysisHandler` objects, to write histograms to disk. If any of these methods are implemented they *must* call the corresponding method of the base class. If e.g. the initialization of an object depends on the prior initialization of another object, the former should call the `init` method of the latter. This ensures that the `doinit` method is only called once per object. Similarly, there is a non-virtual `finish` method for `dofinish`.

`Interfaced` classes should also implement a static `Init` method which will be run once for each class, in which it may create static `InterfaceBase` objects to be used by the `Repository` (see section 4.2.1).

4.1.4 Other classes

Of course, any other class allowed by the standard is allowed in PYTHIA7, although the implementor will manually have to handle the deallocation of objects as well as the input and output to streams, and these classes cannot be handled by the repository. Examples of such classes in PYTHIA7 are `Lorentz5Vector` and `Selector` (see sections 4.4.1 and 4.4.3 respectively).

4.2 The repository

`Repository` is a singleton class which has static lists to keep track of and manipulate all `Interfaced` objects in the setup-phase. There are a number of static functions defined for the manipulation, but there is also a rudimentary command-line interfaced as described in section 3.3.4. A more versatile user interface could be implemented by inheriting from the `Repository` class or by accessing its public static functions from outside.

4.2.1 Interfaces

To manipulate the `Interfaced` objects in the `Repository`, there are a number of interface classes defined e.g. to set and get parameters in an object. These classes all inherit from the `InterfaceBase` class. When an object of such a class is created, it will automatically add itself to the list of interfaces in the `Repository`, hence there should only be one interface object per interface per class. This is conveniently achieved by creating static interface objects in the static `Init()` method of an `Interfaced` class.

4.2.2 Parameters

To create an interface to a parameter of a class, corresponding to e.g. a member variable defined as:

```
class AClass: public Interfaced {
  // ...
private:
  Energy energy;
};
```

the following should be done in the `Init()` method:

```
void AClass::Init() {
  static Parameter<AClass,Energy>
  interfaceEnergy("Energy",
                  "This is the energy in GeV",
                  &AClass::energy, GeV,
                  1.0*GeV, 0.0*GeV, 10.0*GeV);
}
```

This will create an object interfacing the member variable `energy` of the `AClass` class to the `Repository`. The meaning of the arguments to the constructor are as follows: First the name of the parameter used by the repository to identify it, then a short description, followed by a pointer to the actual member, the unit to be used when reading and writing, the default value (1 GeV), the minimum value (0 GeV) and the maximum value (10 GeV).

There are a number of optional arguments which may be given to the `Parameter` constructor: a flag indicating that this parameter may only be inspected (and not changed) by the repository (default=`false`), and a flag to indicate if the parameter always should be limited to within the specified minimum and maximum values (default=`true`). Pointers to member functions for setting the value of the parameter, and to get the current, minimum, maximum and default values, may also be specified in the constructor, in which case the pointer to the actual member may be the null-pointer.

There is also an interface class for vectors of parameters called `ParVector` which can be used for interfacing any container of parameters of a class.

4.2.3 Switches

To interface a switch for selecting different options in a class, the `Switch` class can be used as follows:

```
class AClass: public Interfaced {
```

```
  // ...
private:
  int model;
};

void AClass::Init() {
  static Switch<AClass,int>
  interfaceModel("Model",
                 "Switch between different models",
                 &AClass::model, 1);
  static SwitchOption
  interfaceModelA(interfaceModel, "A", "Use model A", 0);
  static SwitchOption
  interfaceModelB(interfaceModel, "B", "Use model B", 1);
}
```

Here a `switch` object is created given a name, description, a pointer to the actual member variable and a default value. Also a `SwitchOption` object is created for each valid option, and the arguments to the constructor is a reference to the `Switch` object, a name, a short description and the value corresponding to the option.

Just as for the `Parameter` class, it it possible to specify if the switch is read-only, and it is possible to specify pointers to member functions to set and get options rather than using the member variable directly.

4.2.4 Commands

There is a general message interface called `Command`, with which a non-const member functions taking a `std::string` as argument and returning a `std::string` can be made available to the `Repository`:

```
class AClass: public Interfaced {
  // ...
  string doSomething(string);
};

void AClass::Init() {
  static Command<AClass>
  interfaceDoSomething("DoSomething",
                       "Do something interesting",
                       &AClass::doSomething);
```

4.2.5 References between objects

If one `Interfaced` class has a pointer to an object of another `Interfaced` class, this relation can be made available to the `Repository` with a `Reference` object:

```
class AClass: public Interfaced {
  // ...
private:
  Ptr<AnotherClass>::pointer another;
};

void AClass::Init() {
  static Reference<AClass,AnotherClass>
  interfaceAnother("Another",
                   "A reference to another object",
                   &AClass::another);
}
```

As with `Parameter` it is possible to specify that a `Reference` is read-only, and to specify pointers to member functions to set and get the pointer. It is also possible to specify a `rebind` flag indicating that the reference should be rebound when exported to an `EventGenerator` (see section 4.2.6) (default=`true`). It is furthermore possible to specify a flag indicating that the pointer may be assigned the null-pointer (default=`true`). Also, if the pointer is nullable, it is possible to specify a flag – `defnull` – indicating that any null-pointer should be replaced with a pointer to a default object when exported to an `EventGenerator` (default=`false`). Finally it is possible to specify a pointer to a member function which checks if a given object will be accepted if set.

There is also an interface for vectors of references called `RefVector` which can be used for interfacing any container of pointers in a class.

4.2.6 Isolating an event generator

In the setup program, the `Repository` is used to connect different objects with each other, setting their parameters and switches to desired values. In the end the setup program should have built up a complete `EventGenerator` object. There may, however, exist several `EventGenerator` object in the `Repository`, and one object may be pointed to directly or indirectly by several `EventGenerators`. Before an `EventGenerator` can be used it must therefore be *isolated* from the other objects in the repository.

The isolation procedure starts by cloning the specified `EventGenerator` and all (`Interfaced`) objects which are referenced directly or indirectly by the `EventGenerator`. To find out which objects are needed, the `Repository` checks all

`Reference` and `RefVector` interfaces of the `EventGenerator`. The objects found in that way are again checked for `Reference` and `RefVector` interfaces, and so on. If an `Interfaced` class has non-interfaced pointers to other objects, these pointers should be communicated to the `Repository` using the virtual `getReferences()` method (declared in `InterfacedBase`, the base class of `Interfaced`). Note that if the `getReferences()` method is implemented for a class, this implementation must call the `getReferences()` of the base classes.

Cloning the objects is not enough, however, since a cloned object may still be pointing to objects in the repository rather than to their clones. Therefore all pointers must be *rebound* to point to the correct clones. For pointers which are interfaced with `Reference` or `RefVector` (which do not have the `norebind` flag set), this rebinding is automatic. For other pointers this must be done via the
`virtual void rebind(const TranslationMap & trans)`
function (declared in `InterfacedBase`). The argument to this function is a `Rebinder` (via a typedef) and should be used like this:

```
class AClass: public ABaseClass {
  // ...
private:
  Ptr<AnotherClass>::pointer another;

protected:
  virtual void rebind(const TranslationMap & trans)
      throw(RebindException) {
    another = trans[another];
    ABaseClass::rebind(trans);
  }
};
```

Note that if this function is implemented it must call the `rebind` method of the base class.

Finally the `EventGenerator` goes through all reference interfaces of all cloned objects and if one is found which is set to the null-pointer, and if the `defnull` flag is set, and an object of the correct type is available in the `EventGenerators` list of default objects, that object will be assigned to the interface.

After an `EventGenerator` has been isolated, all the objects that were cloned and included in the run will have a pointer to this `EventGenerator`. Though this pointer an `Interfaced` object will have access to global properties of the run. It is important to note the difference between the setup phase and the run phase in this respect. If, during the setup phase, an object wants to access a `ParticleData` object it should do so using static methods of the `Repository`, while during the run phase it should use its `EventGenerator` pointer and use the corresponding non-static members of the `EventGenerator`.

4.3 Creating new handler classes

When implementing a physics model in Pythia7, the procedure is to create a new class inheriting from one of the existing *handler* classes, overriding the relevant virtual functions. Since all handler classes inherits from Interfaced which in turn is persistent, there are a number of common things to do for any new handler class AClass:

- The class must have a public default constructor.
- Exactly one object of either class ClassDescription<AClass>, AbstractClassDescription<AClass>, NoPIOClassDescription<AClass> or AbstractNoPIOClassDescription<AClass> must be instantiated to register this class with the persistent stream classes. This is typically achieved by a static member. Which of the description classes to choose depends on whether AClass has members which must be written persistently and whether it is abstract or concrete.
- The templated ClassTraits should be specialized to supply information about this class to the persistent streams. At least the static ClassTraits<AClass>::className() method must be overridden to return a unique and platform-independent string representing the name of AClass. This is conveniently done using the PYTHIA7_DECLARE_CLASS_TRAITS macro.
- The templated BaseClassTrait must be specialized for each immediate base class which is persistent so that BaseClassTrait<AClass,1>::NthBase is a typedef for the first base class, BaseClassTrait<AClass,2>::NthBase a typedef for the second, and so on. This is needed by the persistent streams and is conveniently done using the PYTHIA7_DECLARE_CLASS_TRAITS macro.
- If the class has members which need to be written and read persistently, the (non-virtual) methods
  void persistentOutput(PersistentOStream &) const and
  void persistentInput(PersistentIStream &, int)
  must be implemented. (The names of these functions can be controlled in the ClassTraits class.)
- If the class is concrete it must implement a clone method as described in section 4.1.3.
- If the object produced from a clone() call needs some initialization to be useful in the Repository, the fullclone() should be implemented to return an initialized object.
- If the objects of this class needs to be initialized before an EventGenerator run is started, the void doinit() throw (InitException) should be implemented (which must always must call the doinit() method of the base classes). If the initialization fails, the method should throw an InitException.

- If the objects of this class needs to do something after an `EventGenerator` run, e.g. write out statistics, the `void dofinish()` method should be implemented (which must call the `dofinish()` method of the base classes).
- The class must implement a static `Init()` method, possibly containing static `InterfaceBase` objects as described in section 4.2. This method is called once for each class in the construction of the static `ClassDescription` object of that class.
- If the class has pointers to other objects which are not interfaced, the method `vector<Ptr<InterfacedBase>::pointer> getReferences()` must be implemented. The returned vector should contain the result from the call to `getReferences()` of the base class, plus all non-interfaced pointers in `AClass`.
- If the class has pointers to other objects which are not interfaced, the method `void rebind(const TranslationMap & trans)` must be implemented, calling the `rebind()` method of the base class using the argument as described in section 4.2.6.

This may seem like a lengthy and cumbersome procedure just to create one new class. But typically only a few of these items are necessary, and most things can be generated automatically. PYTHIA7 contains a number emacs-lisp functions (in the `Templates` directory) for easy generation of complete skeletons for new classes, containing default versions of the methods needed.

4.3.1 Partonic cross-sections

The `PartonXSecFn` class is the handler class from which all classes implementing hard $2 \rightarrow n$ partonic cross sections should be inherited. In the basic form a `PartonXSecFn` should represent the function $\hat{\sigma}_{ij\rightarrow X}(\hat{s})$ so that the total integrated cross section for the corresponding sub-process is given by:

$$\sigma = \int \frac{dx_1}{x_1}\frac{dx_2}{x_2} xf_{1,i}(x_1) xf_{2,j}(x_2) \hat{\sigma}_{ij\rightarrow X}(S_{\max} x_1 x_2), \tag{4}$$

where x_1 is the momentum fraction of parton i and xf_{1i} is the corresponding momentum density (and similarly for 2 and j).

But `PartonXSecFn` is more than a simple function and to be usable in the sub-process selection described in section 3.3.2, there are a number of different virtual member functions which can be specialized.

- `sigHatMax(const SInterval & S, const SInterval & SHat,`
`const cPDPair &, const KinematicalCuts &) const;`
should return vector of function objects of class `SigmaHatMaxBase` corresponding to $\hat{\sigma}^{\max}$ in eq. 1. Each of these objects should correspond to a term with a specific momentum and colour geometry (see below). The arguments are an interval in S, an interval in $\hat{s}$, the pair of incoming partons and a `KinematicalCuts` object. This method is called in the initialization phase and it should be noted that it is a `const` function, since the same object may be be used for several different sub-processes. Any process-dependent initialization should therefore be stored in the returned `SigmaHatMaxBase` objects.

- `pdfScale();`
  is called after a phase-space point has been generated by the `CollisionHandler`. It should return the scale of the hard sub-process to be used in the parton density functions. No arguments are given, but since `PartonXSecFn` inherits from the `LastXCombInfo` class, all information about the selected phase-space point is already available. The `PartonXSecFn` typically needs to generate internal degrees of freedom to be able to calculate the scale, and these should then be saved for subsequent calls.

- `weight();`
  should return $\hat{\sigma}(\hat{s})/\hat{\sigma}^{\max}(\hat{s})$, the ratio between the exact cross section function and the approximate upper limit at the generated phase-space point. This ratio should be less than unity. But occasional weights above one can be handled by a compensation mechanism in the `CollisionHandler`. If the approximate cross section is the result of an integral over internal degrees of freedom, $\hat{\sigma}^{\max} = \int dz_1 \cdots dz_n d\hat{\sigma}^{\max}(\hat{s}, z_1, \ldots, z_n)/dz_1 \cdots dz_n$, this function may generate some or all of these degrees of freedom and return the weight
  $$w = \frac{\frac{d\hat{\sigma}(\hat{s},z_1,\ldots,z_n)}{dz_1 \cdots dz_n}}{\frac{d\hat{\sigma}^{\max}(\hat{s},z_1,\ldots,z_n)}{dz_1 \cdots dz_n}}. \qquad (5)$$
  Again, no arguments are needed. If internal degrees of freedom are generated, these should be saved for future use.

- `construct(tSubProPtr sub);`
  is called if the selected phase-space point has been accepted by the `CollisionHandler`, and takes a pointer to a `SubProcess` as argument. In this function the complete hard parton–parton sub-process should be constructed using the internal degrees of freedom previously generated in `pdfScale` and/or `weight` together with the remaining ones that need to be generated here.

The momentum geometry for a general $2 \to n$ process is described in terms of a vector of $n-1$ sets of four integers, each set corresponding to one of the internal lines (a four-parton vertex is treated as an internal line in this respect). The first two numbers corresponds to the indices of the partons of one of the vertices, the last two to the other. The incoming partons are numbered 1 and 2, the outgoing $3 \ldots n+2$, and the internal lines $n+3 \ldots 2n+1$. This geometry is in most cases completely unphysical – it is not possible to differentiate between different Feynmann diagrams with the same final state – and can be left out. However, especially if a $2 \to n$ sub-process is followed by a partonic cascade, information about which geometry is the most likely may be necessary.

The colour geometry is given as a vector of $2+n$ numbers corresponding to the flow of colour from each of the incoming and outgoing partons. Also so-called colour-junctions connecting three colour triplets (or three anti-colour triplets) to each other, can be described by specifying (anti-) colour flow to an imaginary parton with index $> 2+n$.

`PartonXSecFn` is the most general form for a hard $2 \to n$ partonic cross-section. Simple $2 \to 1$ sub-processes are easy to implement using this base class, although the handling of resonances with the *width generator* mentioned in section 3.2 has not yet been implemented properly. For $2 \to 2$ processes there is a special base class called `THatXSecFn` inheriting from `PartonXSecFn` and representing $d\hat{\sigma}(\hat{s},\hat{t})/d\hat{t}$.

The virtual functions which needs to be implemented in `THatXSecFn` are as follows.

- `sigHatMax(const SInterval & S, const SInterval & SHat,`
  `const cPDPair &, const KinematicalCuts &) const;`
  should now return `THatFunctionBase` objects which inherits from the `SigmaHatMaxBase`. This class automatically takes care of the integral over and generation of $\hat{t}$ given any $d\hat{\sigma}(\hat{s},\hat{t})/d\hat{t}$ function (although special integration and generation of course may be implemented in derived classes).

- `getOutgoing() const;`
  should return the types of the pair of outgoing partons for the selected phase-space point and colour and momentum geometry.

- `getIntermediate() const;`
  should return the exchanged parton for the selected pair of outgoing partons, the selected phase-space point and colour and momentum geometry.

- `reweight();`
  is called by the `weight` function to give any weight related to approximations made when giving the `THatFunctionBase` objects in the `sigHatMax` method. One example is to return the ratio between the α_S at the scale of the chosen phase-space point and the maximum value which may have been used in the initialization.

A class inheriting from `THatXSecFn` may, of course, also override other virtual functions of `PartonXSecFn` if needed.

A general base class for $2 \to n$ processes, where only the squared matrix elements for different colour and momentum geometries need to be implemented will be provided in the future.

4.3.2 PDF's and remnant handling

All parton density functions should be implemented in classes inheriting from `PDF-Base`. Just as for `PartonXSecFn`, the `PDFBase` class should implement both an approximate upper limits of the density functions for each separate parton in a particle, as well as the functions themselves. The approximate upper limits shall be given as `ApproxPDF` objects, each of which represents a sum of powers of $\log(1/x)$, the logarithmic momentum fraction:

$$xf^{\max} = \sum_i c_i \log(\frac{1}{x})^{a_i}. \tag{6}$$

For a given x, this function should be larger than the true function for any scale allowed by the kinematical cuts. The `PDFBase` class has methods to automatically produce such approximations, but derived classes may specify their own.

Also the momentum distribution of the incoming particles should be approximated by `ApproxPDF`s, making it simple to convolute with these. Even in complicated cases such as distribution of partons in a pomeron in a photon in an electron with varying momentum, it is easy to convolute `ApproxPDF`s to get the total parton momentum densities:

$$xf_{\mathrm{tot}}^{\max}(x) = \int_x^1 \frac{dx'}{x'} x' f_1^{\max}(x') \frac{x}{x'} f_2^{\max}(\frac{x}{x'}) = \sum_{i,j} c_{1i} \cdot c_{2j} \log(\frac{1}{x})^{a_{1i}+a_{2j}+1}. \tag{7}$$

The important virtual methods of `PDFBase` are as follows:

- `canHandleParticle(tcPDPtr particle) const;`
  should return `true` only if the implemented PDF can be used for the particle type given in the argument.

- `partons(tcPDPtr particle) const;`
  should return a vector of partons which may be extracted from the particle type given in the argument.

- `approx(tcPDPtr particle, const PDFCuts &) const;`
  given a particle and an object representing the ranges in momentum fraction and scale where this PDF will be used, this method should return a map of `ApproxPDF` objects indexed by the corresponding parton type.

- `xfx(tcPDPtr particle, tcPDPtr parton, Energy2 scale,`
  `double x, double eps = 0.0, Energy2 offshell = 0.0*GeV2) const;`
  should return the momentum density, given as arguments the particle, the parton, the scale at which the parton is resolved, the momentum fraction x of the parton and, optionally, `eps`$= 1 - x$ (for precision reasons when x is close to 1) and the offshellness of the particle (mainly intended for resolved virtual photons).

- `xfvx(tcPDPtr particle, tcPDPtr parton, Energy2 scale, double x, double eps = 0.0, Energy2 offshell = 0.0*GeV2) const;` may be implemented in the same way as `xfx` to give only the valence part of the momentum density. The default version simply returns 0.

`PDFBase` has a pointer to a `RemnantHandler`, which should be able to construct the remnants for any parton which may be extracted from a given particle. `RemnantHandler` has the following virtual functions:

- `canHandle(tcPDPtr particle, const cPDVector & partons) const;` should return `true` if the `RemnantHandler` can handle the remnant generation for a given particle type and a vector of partons to be extracted.

- `getRemnants(tcPDPtr particle, tcPDPtr parton, double x, Energy2 sMax, TransverseMomentum & kt)` should return a vector of remnants. The arguments are the particle, the extracted parton, the momentum fraction and an upper limit on the invariant mass squared of the remnants. The remnants should be returned in their own center-of-mass system assuming that the incoming particle is along the positive z-axis. The intrinsic transverse momentum of the extracted parton may also be returned through a reference argument.

A second version of the `getRemnants` will be supplied in the future, implementing multiple parton extractions from a particle.

The whole process of extracting partons from particles and creating remnants is administered by a `PartonExtractor` object, which has two main virtual functions:

- `weight();` should return the product of the ratios of the true to the approximate parton density functions used for the selected phase-space point. The function takes no argument and all information about the selected phase-space point can be obtained from the `LastXCombInfo` base class.

- `construct(tCollPtr, tStepPtr, tSubProPtr)` given pointers to the current `Collision`, the first `Step` of that collision and the `SubProcess`, the complete kinematics of the hard sub-process should be constructed and be put into the `Step`.

The `PartonExtractor` also has a number of methods to be used by subsequent step handlers to access information about the parton densities and remnant handlers.

4.3.3 Luminosity functions

The `LuminosityFunction` base class should be used to describe the momentum distribution of the beam particles. As for the parton densities, these should be given both as an approximate upper limit in the form of `ApproxPDF` objects and as a member function giving the exact distribution. The following virtual methods should be implemented by a derived class:

- `canHandle(const cPDPair &) const;`
  should return true if the class can handle the pair of incoming particles given as arguments.

- `getSBins(const cPDPair &) const;`
  should return a vector of intervals in total particle–particle invariant mass squared, S, for which this class can be used.

- `probDists(const SInterval &) const;`
  should return a pair of `ApproxPDF`s giving the approximate upper limits of the momentum distributions of the incoming particles for the interval in S given as argument.

- `weight();`
  for the selected phase-space point (which is available through the `LastXComb-Info` base class) return the product of the ratios of the true to the approximate momentum distribution functions.

4.3.4 Step handlers

After the initial step of a `Collision` has been generated, a sequence of `Step-Handler`s are called to perform steps necessary to complete the generation of one collision. The `StepHandler` base class is used for any kind of such process and has only one virtual method to be overridden by derived classes:

- `handle(PartialCollisionHandler & ch, const tcPVector & tagged,`
  `const Hint & hint);`
  is given a reference to the current `PartialCollisionHandler`, a vector of *tagged* particles and a `Hint`. The idea is that the `StepHandler` should examine the tagged particles and the hint to see if there is anything that it can do. If so, it should do whatever it is supposed to do and, if this results in new particles being produced or old ones being modified, it should get a copy of the last `Step` by calling the `newStep()` method of the `PartialCollision-Handler`, and put the resulting particles there.

Although only the tagged particles are directly available in the `handle` method, the `StepHandler` is free to manipulate any part of the event produced so far which is available from the `PartialCollisionHandler` using the `currentEvent()` method.

The basic `Hint` class simply contains a list of tagged particles and a scale. The list of particles in the `Hint` is almost the same as the ones passed as argument to `handle`, but the latter only contains particles which are still present in the last

`Step`, when `handle` is called. The meaning of the scale in the `Hint` is unspecified – the implementor of a `handle` method should clearly document how this scale is interpreted. A `StepHandler` class can have tailor-made `Hint` classes, in which case the `handle` method should `dynamic_cast` the `Hint` supplied in the argument. Note, however, that this cast must be checked manually since there is no guarantee that the hint is of the expected class.

A special `Hint` object is the *default* hint, obtainable from the static `Default` method in the `Hint` class, and corresponds to telling the `StepHandler` *Look through the current event, check if there is anything to do and do it.*

Note that a `StepHandler` does not need to do anything. In fact, also things like analysis and different kinds of cuts can be implemented as a `StepHandler`. In the latter case, the `StepHandler` can look if the event would fail some cuts, in which case it should throw a `Veto` exception - causing the `EventHandler` to throw away the current event and generate a new one. Alternatively, the `StepHandler` may throw a `Stop` exception, in which case the `EventHandler` will stop generating and return the event generated so far to the calling program.

Any one can at any time during the generation of an event add a `StepHandler` and a `Hint` to a `PartialCollisionHandler` using the `addStep` method. Here one should also specify to which group of handlers the `StepHandler` should be added and at which level in that group. The group can be given as `Group::subpro`, `Group::cascade`, `Group::hadron` or `Group::decay` and the level can be given as `Group::before`, `Group::main` or `Group::after`. Specifying e.g. `Group::cascade` and `Group::before` (`Group::after`) will add a `StepHandler` and a `Hint` to the list of handlers to be called before (after) calling the `CascadeHandler`, while specifying `Group::cascade` and `Group::main` will replace the current `CascadeHandler` (if not the null-pointer is specified) and add the `Hint` to the list of hints for which the main `CascadeHandler` will be called. Note that if the cascading has already been performed, the `PartialCollisionHandler` will restart the cascading in the next step.

When the generation of an event is started, the groups of step handlers will be filled with handlers which are given *default* hints. These handlers are specified in the setup phase in the `PartialCollisionHandler`. It is also possible to specify handlers in the different `SubProcessHandlers`. The default handlers specified in the `SubProcessHandler` selected for an event will take precedence over the ones in the `PartialCollisionHandler`.

The groups are then processed in order, first the list of post-sub-process handlers[5], then the list of pre-cascade handlers, followed by the main `CascadeHandler` possibly called several times with different hints, then the post-cascade handlers, continuing in the same way with the hadronization and decay groups.

[5]Note that there are no pre-sub-process handlers and that the main sub-process is handled outside of this structure

4.3.5 Cascade handlers

So far no model for QCD cascades has been implemented in Pythia7, and the CascadeHandler class currently does not introduce any functionality beyond the StepHandler base class.

4.3.6 Hadronization handlers

The Lund string fragmentation model was the first substantial physics module to be added to Pythia7. The implementation is discussed briefly in section 5 and in detail in ref. [8].

4.3.7 Decay handlers

The DecayHandler should be used to administer the decay of unstable particles. The actual generation of phase space for decay products is handled by the Decayer assigned to each decay channel. The DecayHandler base class implements the handle method declared in StepHandler, where it goes through the tagged particles and, for the unstable ones, selects a decay channel, asks the Decayer to perform the actual decay and inserts the children in the new Step, setting up mother–daughter relationships, etc. Children which are unstable are again decayed, until only stable particles are left.

Note that the decay of an unstable particle may also be done inside other step handlers using the Decayer objects given in the decay table of a ParticleData object. This is useful e.g. in a CascadeHandler where the decay of a top quark may be required in the middle of the cascade.

4.3.8 The Standard Model

The EventGenerator has a pointer to a StandardModelBase object which carries information about the Standard Model parameters to be used in a run. StandardModelBase has inlined functions to access information which can be calculated at initialization time. To calculate some parameters, StandardModelBase has pointers to objects implementing α_{EM}, α_{S} and the CKM matrix using the AlphaEMBase, AlphaSBase and CKMBase classes respectively.

So far, there is no *beyond* Standard Model physics in Pythia7. But it is natural to implement e.g. SUSY parameters in classes inheriting from StandardModelBase. Handler classes which need access to SUSY parameters, would then dynamically cast the StandardModelBase object available through the EventGenerator to a SUSY parameter object to access these parameters.

4.3.9 Random Numbers

The `EventGenerator` has a pointer to a `RandomGenerator` object which can be used by any `Interfaced` object through the inlined `rnd()` method returning a random number in the range $]0,1[$. Also methods for getting random numbers in other ranges, as well as random integers are given.

`RandomGenerator` only defines the interface to a random generator and assumes that derived classes will handle classes derived from the `RandomEngine` class of CLHEP. A derived class need only implement the virtual `randomGenerator` method returning a reference to the `RandomEngine` object.

Generating random numbers is a very central part in event generation and typically 10-20% of the total running time is spent generating random numbers. To reduce the overhead of having to call virtual functions every time a random number is needed, the `RandomGenerator` uses the `flatArray` method of the `RandomEngine` object to generate a large number of numbers which are cached in a large vector, In this way an inlined member function of `RandomGenerator` can be used to get a random number, and only once every thousand calls or so, a virtual method needs to be called to fill the cache.

4.3.10 Exceptions

The `Exception` class should be used for all exceptions thrown by a handler which are supposed to be caught by either the `Repository` in the setup phase or by the `EventGenerator` in the running phase. A class deriving from `Exception` should, in the constructor, specify the kind of error which has occurred using the `severity()` member function and write a message to the protected `theMessage` member variable of type `std::ostringstream`.

The error types are enumerated as follows:

`unknown`
: if no type was specified, the program will abort and dump core as soon as the exception is thrown.

`info`
: this is not really an error and the exception should in principle not be thrown. Instead it should be logged using the `logException` method of the `EventGenerator` class which simply writes out the message to a log file. It is possible to fix the maximum number of messages of the same kind which will be written out.

`warning`
: should be treated the same way as the `info` exception, i.e. it should be logged with the `EventGenerator`.

`eventerror`
: the generation of the current event will be aborted. The exception will be caught by the `EventGenerator` which may decide to abort the execution if too many exceptions of a given type is caught.

`runerror`
: the whole run will be aborted but the exception is caught by the `EventGenerator` who gracefully returns the control of the execution to the calling process.

`maybeabort`
: a serious error has been found which may depend on a programming error. `EventGenerator` catches this exception to write out a message but then re-throws the error so that the calling function may catch it and continue executing.

`abortnow`
: the message is written directly to `std::cerr` and the process is aborted and a core dump is produced before the throwing procedure has started the stack unwinding.

In the end of each run, a summary of all exceptions which have occurred is given in the log-file (class name and count).

4.4 Utility classes

PYTHIA7 contains a number of small utility classes, and surely the number will increase as more physics models are implemented in the framework. Some of these classes are not specific to PYTHIA7, and they should maybe migrate into CLHEP, others are more specific. Here are some examples:

4.4.1 Lorentz5Vector

`Lorentz5Vector` inherits from the `LorentzVector` class of CLHEP, and simply adds a data member implementing the mass component. The reason is not just to cache the mass component for easy access, it is also useful to avoid precision problems [6] and in general it can be technically convenient to have a mass component different from the invariant mass.

[6] even if double precision is used, a TeV neutrino may e.g. have unacceptably large error in its invariant mass

4.4.2 Math functions

Inside the namespace `Pythia7::Math` there are a number of mathematical functions and classes defined. Here are some examples:

- `gamma(double), lngamma(double)`
  returns the (log of the) gamma function.
- `exp1m(double)`
  returns $1 - \exp(x)$ to highest possible precision for x close to 0.
- `log1m(double)`
  returns $\log(1 - x)$ to highest possible precision for x close to 0.
- `template <int N> double Pow(double)`
  returns the argument raised to the integer power given as the template argument.

4.4.3 Selector

The `Selector` class declared as

```
template <typename T, typename WeightType = double>
class Selector;
```

stores objects of type `T` and associates them with a weight. Using the `select` method, giving random numbers in the range $]0, 1[$, objects are retrieved one at the time with a probability proportional to their weight.

4.4.4 SimplePhaseSpace

The `SimplePhaseSpace` class has a number of static member functions to distribute two or three particles in phase space in their center of mass system. The methods are called `CMS` and take two or three references to particles and a total invariant mass squared as argument, together with angles and, in the case of three particles, energy fractions. In some cases the angles may be generated isotropically given an `RandomGenerator` as argument.

4.4.5 Other utility classes

- `Interval` is a templated class representing an interval of numbers corresponding to the template argument type. E.g.
  `Interval<double> interval(2.2, 13.0)` represents the interval $[2.2, 13.0[$

- `HoldFlag` is a templated class used to give temporary values to a variable in an exception-safe way such that the old value is recovered when the `HoldFlag` object is destroyed.
- `Triplet` is a templated class completely analogous to the standard `std::pair` class.

4.5 Documentation

Besides this document, the main documentation of PYTHIA7 is found in the header files. These are commented in a way such that they can be automatically converted into HTML pages using a small *awk* script included in the distribution.

For the handler classes there is also some documentation provided in the definition of the interfaced parameters and switches. In the future it should be possible to extract this information to produce a brief description of an `Interfaced` class and its interfaces in HTML and LaTeX format.

For handler classes implementing a specific physics model, it is possible to communicate information about this model using the virtual `modelDescription` and `modelReferences` functions defined in the `Interfaced` class.

- `modelDescription();`
  should return an `std::string` containing a LaTeX `\item` with a brief description of the model.
- `modelReferences();`
  If `modelDescription` returns a text with citations, the corresponding `\bibitem`s should be returned here.

If a class implements any of these two functions the corresponding methods in the base class should also be called and the result concatenated. After an `EventGenerator` run, the `modelDescription` and `modelReferences` methods of all objects which has been used[7] will be called and the result will be put in a `.tex` file from which a user can cut and paste the relevant references into a publication where results from PYTHIA7 has been used.

5 Case study : The Lund String Fragmentation

In this section, we briefly describe the implementation of the Lund fragmentation model that will be the main option for hadronization in PYTHIA7. We focus on the design analysis and present the key components of its implementation. A complete description of the analysis and design is given in [8].

[7] To indicate that an object has actually been used, the `useMe()` method defined in the `Interfaced` class should be called at least once.

5.1 The Lund fragmentation algorithm

In the Lund model, the colour field between two colour-connected partons is approximated by a massless relativistic string, which can break by the creation of $q\bar{q}$ pairs to produce hadrons (see [10, 11] for a complete review on the Lund model).

In the case of generic multiparton states, such as those produced e.g. by parton showering in e^+e^- annihilation, the string is stretched from the quark to the antiquark endpoints via the colour-connected gluons, which can be considered to be internal excitations of the string field. The mechanism describing the fragmentation of such a system is very complicated and presented in detail in [12, 13]. In this section, we briefly introduce the main features of the Lund algorithm that will lead to the class category identification.

For a Monte Carlo implementation, the fragmentation algorithm is conveniently expressed in the momentum space representation, where it can be formulated as an iterative sequence of steps starting from both *endpoints* and proceeding towards the middle of the string [12]. Each step (i) is taken from the *last* endpoint, previously produced in the same direction, to a new production vertex point where a new $q_i\bar{q}_i$ pair is created. The flavour $\bar{q}_{i-1}$ left at the last endpoint then pair off with the new q_i to form the new hadron. The left-over flavour $\bar{q}_i$ then form the new starting endpoint for the next step. These iterations proceed until the remaining mass of the string is below a minimum value, at which point two final hadrons are produced.

5.2 The design analysis

The current design of the Lund fragmentation modules is structured around two main class categories :

The string representation : Encapsulates the string information and provides a flexible framework for the fragmentation algorithm to operate on. The main strategy used for the design development was to decouple the category responsible for the string fragmentation administration from any particular string representation. Classes using a string object should not have explicit knowledge of its internal structure but rather access information through well-defined interfaces.

The fragmentation administration : To mirror the analysis of section 5.1, the fragmentation management should results in a step-by-step updating procedure of endpoints, each step corresponding to a new hadron creation. This is achieved by the development of an `EndPoint` type used by the fragmentation administrator (the `LundFragHandler`) to manage the sequence of steps. To allow for improved code maintainability and flexibility in terms of new physics developments, the `LundFragHandler` should not depend on the `EndPoint` implementation and should use it as an “iterator-like” type in the management of the iterative procedure.

5.3 The `LundFragHandler` class

The `LundFragHandler` class is responsible for the administration of the fragmentation procedure. The main components are :

- A `String` object that holds all information about the current string to hadronize.
- Three `EndPoints` : to describe the last two endpoints previously produced from the ends of the string, and the new endpoint of the current step being performed. Each `EndPoint` encapsulates information about the endpoint parton type and the breakup vertex position.
- Pointers to the `LundPtGenerator`, `LundZGenerator`, `LundFlavourGenerator` objects for the p_t, z and flavour generation.
- A list of `Hadron`s holding information on the newly created hadrons.
  In the process of fragmenting the `String`, the `LundFragHandler` will often try several times to complete the hadronization. To avoid the extra overhead of creating and destroying `Particle` objects, `Hadron` provides a simplified implementation which contains the type and momentum of a temporary particle.

To make the `LundFragHandler` class reusable, we split the string-breaking administration into separate methods.

- `handle(PartialCollisionHandler & ch, const tcPVector & tagged, const Hint & hint);`
  This is the only public method of the `LundFragHandler` and it overrides the one inherited from the `HadronizationHandler` base class (see sections 4.3.4 and 4.3.6). Its task is to look through the list of *tagged* particles in the current `Step`, extract the strings if any, and send them to the `hadronize` method to be fragmented. At the end of the fragmentation, it gets from the `CollisionHandler` a copy of the last `Step`, in order to add to the event record the newly created particles before informing the `CollisionHandler` to continue the generation. This procedure is very general and will probably be moved to the `handle` method of the `HadronizationHandler` class.
- `Hadronize(const tPVector &);`
  Is the major method in the string fragmentation administration. Given a vector of particles, received from the `handle` method, it returns the list of created hadrons. Its administration task can be summarized as follows :

1. Send the incoming particles to the `initHadronization` method to initialize the fragmentation.
2. Ask the `getHadron` to administrate the creation of a new hadron.
3. Re-invoke (2) while the remaining energy of the `String` is above a minimum value, given by the `Wmin2` method, otherwise ask the `finalTwoHadron` method to produce the last two hadrons in the fragmentation process.

- `initHadronization(const tPVector &);`
  Is the method that create the `String` object of the `LundFragHandler`. Given the particles, coming from `Hadronize`, it has the responsibility to find the string type (*closed* or *open* string) and to call either the `initOpenString` or the `initClosedString` method that selects the correct initial particle to be sent to the `String` constructor. Besides that, it initializes the `LundFragHandler` variables and prepares the iterative process by setting the correct rightmost and leftmost `EndPoint`s of the string.

- `getHadron();`
  Administrates the creation of a new hadron. Its main task is :

  1 Generate the new `EndPoint` in the current step. For that `getHadron` makes use of the different `LundGenerators`. Try to produce a new hadron joining the current and the last `EndPoint`.

  2 If a solution is found then the full hadron kinematics is computed, otherwise it delegates to the `stepping` method the responsibility to find the correct breakup position that corresponds to a physically acceptable solution for the selected kinematics.

  3 Invoke the `loopBack` method that properly updates the `String` before starting a new step.

5.4 Outlook and comparison with Pythia version 6

The first implementation of the Pythia7 fragmentation modules provides functionalities comparable to the default hadronization option of Pythia version 6.1. However, some functionality, such as the collapsing of low-mass strings into one or two particles, and the treatment of so-called junction strings, is still missing.

The design and implementation of physics models for p_t, z and flavour generation have been completed, up to the level of the Pythia6.1 functionality, for the `LundPtGenerator` and `LundZGenerator`. The current version of the `LundFlavourGenerator` implements the whole meson production model and the popcorn model for baryon production. An open structure to handle the flavour generation in the fragmentation phase has been provided for the foreseen implementation of the modified popcorn model [14].

Comparisons between the PYTHIA7 fragmentation modules and PYTHIA version 6 have been carried out at the level of test-case per class and at the level of test-bench for the whole hadronization phase. Excellent agreement between the two simulation results has been found.

6 Bugs, lacking features and future plans

The current PYTHIA7 is a proof-of-concept version and it should certainly not be assumed to be bug-free. In fact it should not even be assumed to be able to produce anything particularly useful. But is should be stable enough to start implementing real physics models. In any case, anyone is more than welcome to play around with the code and give suggestions or point out bugs to `leif@thep.lu.se`.

Most of what is missing in the current version of PYTHIA7 is physics models, while most of the underlying structure where such models can be implemented is in place. There are, however, some exceptions where the current structure most likely needs to modified.

One such case is multiple interactions. Here the remnant handlers need to be modified in order to allow for extraction of several partons from a particle. The actual administration could be handled by a specialized `PartonExtractor` class, but one could also imagine introducing a new group of step handlers.

For hard parton–parton scatterings, a new base class should be introduced where it is possible to specify a general $2 \to n$ matrix element.

PYTHIA7 is prepared for interfacing to the SIunits package[15] for automatic compile-time checking of dimensions of physical variables. To facilitate the transition to the SIunits package, all dimensionful variables uses the `typedef`s of `double` defined in the `Units.h` file and all dimensionful constants are multiplied with units as defined in the `SystemOfUnits.h` file of CLHEP. This means that variables are handled as follows:

```
Energy e = 2.5*GeV;
Energy2 s = e*e;
Energy m = sqrt(s);
```

The user interface also needs to be improved. It is our hope that someone else would take on the challenge of writing a cool graphical user interface. This should not be too difficult as everything important for a user interface is available through the `Repository` via the interface classes.

References

[1] Further information on the Pythia7 project will be available at `http://www.thep.lu.se/Pythia7/` See also L. Lönnblad, *Comput. Phys. Commun.* **118** (1999) 213.

[2] G. Cosmo, S. Giani, N. Hoimyr et al., "GEANT 4 : an object-oriented toolkit for simulation in HEP", CERN LHCC 95-70, LCRB Status Report RD 44, S. Giani et al., CERN-LHCC-97-40. See also `http://wwwinfo.cern.ch/asd/geant/geant4.html`

[3] L. Lönnblad, Proceedings of the MC93 International Conference on Monte Carlo Simulations in High Energy and Nuclear Physics, Febuary 1993, Tallahassee, Florida, USA, Eds. P. Dragovitsch et al., p. 202.

[4] L. Lönnblad, *Comput. Phys. Commun.* **84** (1994) 307. See also `http://wwwinfo.cern.ch/asd/lhc++/clhep/index.html`

[5] R. Kuhn, et al., 'APACIC++, A PArton Cascade in C++, version 1.0', hep-ph/0004270.

[6] M. Dobbs, J.B. Hansen, 'HEPMC, a C++ Event Record for Monte Carlo Generators', see `http://mdobbs.home.cern.ch/mdobbs/HepMC/`

[7] L. Garren, 'StdHep 4.08 Monte Carlo Standardization at FNAL', Fermilab PM0091. See also `http://www-pat.fnal.gov/stdhep/c++/`

[8] M. Bertini, The Lund String Fragmentation in Pythia version 7, LU-TP 00-24.

[9] C. Caso et al, *European Physical Journal* **C3** (1998) 1. See also `http://pdg.lbl.gov/`.

[10] B. Andersson, G. Gustafson, G. Ingelman and T. Sjöstrand, *Phys. Rep.* **97** (1983) 31.

[11] B. Andersson "The Lund Model", Cambridge University Press, 1998.

[12] T. Sjöstrand, *Nucl. Phys.* **B248** (1984) 469

[13] T. Sjöstrand, *Comput. Phys. Comm.* **82** (1994) 74. See also `http://www.thep.lu.se/staff/torbjorn/Pythia.html`

[14] P. Edén and G. Gustafson, *Z. Phys* **C75** (1997) 41.

[15] W. Brown, 'Introduction to the SI library of unit based computation', FERMILAB-CONF-98-328, October 1999. See also `http://www.fnal.gov/docs/working-groups/fpcltf/Pkg/SIunits/doc/0SIunits.html.gz`