

HepMC 2

a C++ Event Record for Monte Carlo Generators

<http://savannah.cern.ch/projects/hepmc/>

User Manual Version 2.03

February 4, 2008

Matt Dobbs

University of Victoria, Canada

Jørgen Beck Hansen

CERN

Lynn Garren

Fermi National Accelerator Laboratory

Lars Sonnenschein

CERN

Abstract

The HepMC package is an object oriented event record written in C++ for High Energy Physics Monte Carlo Generators. Many extensions from HEPEVT, the Fortran HEP standard, are supported: the number of entries is unlimited, spin density matrices can be stored with each vertex, flow patterns (such as color) can be stored and traced, integers representing random number generator states can be stored, and an arbitrary number of event weights can be included. Particles and vertices are kept separate in a graph structure, physically similar to a physics event. The added information supports the modularisation of event generators. The package has been kept as simple as possible with minimal internal/external dependencies. Event information is accessed by means of iterators supplied with the package.

HepMC 2 is an extension to the original HepMC written by Matt Dobbs.

Contents

1	Introduction	2
1.1	Features of the HepMC Event Record	3
2	HepMC 2	3
2.1	Overview of Changes Since HepMC 1.26	4
3	Package Structure	4
3.1	Dependencies	5
3.2	Namespace	5
3.3	Performance	5
4	Overview of Core Classes	6
4.1	HepMC::GenEvent	6
4.1.1	HepMC::PdfInfo	7
4.1.2	HepMC::HeavyIon	7
4.2	HepMC::GenVertex	7
4.3	HepMC::WeightContainer	8
4.4	HepMC::GenParticle	8
4.4.1	HepMC::Flow	9
4.4.2	HepMC::Polarization	10
4.4.3	HepMC::FourVector	10
4.5	HepMC::IO_BaseClass	10
5	Overview of Iterators	11
5.1	HepMC::GenEvent::vertex_iterator	11
5.2	HepMC::GenEvent::vertex_const_iterator	11
5.3	HepMC::GenEvent::particle_iterator	11
5.4	HepMC::GenEvent::particle_const_iterator	11
5.5	HepMC::GenVertex::vertex_iterator	11
5.6	HepMC::GenVertex::particle_iterator	12
6	Building HepMC	13
7	Examples	13
8	Deprecated Classes	13
8.1	HepMC::IO_Ascii (deprecated)	14
8.2	HepMC::ParticleData (deprecated)	14
8.3	HepMC::ParticleDataTable (deprecated)	14
9	Acknowledgements	15

1 Introduction

This user manual is intended as a companion to the online documentation¹, and together with the examples should provide a friendly introduction to the HepMC event record. A general overview is available in Ref. [1].

The HEP community has moved towards Object-Oriented computing tools (usually C++): most upcoming experiments are choosing OO software architecture, and Pythia 8 [2] and Herwig++ [3], written in C++, are available. A standard event record must be simple for the end user to access information, while maintaining the power and flexibility offered by OO design. The HepMC event record has been developed to satisfy these criteria.

HepMC is an object oriented event record written in C++ for Monte Carlo Generators in High Energy Physics. It has been developed independent of a particular experiment or event generator. It is intended to serve as both a “container class” for storing events after generation and also as a “framework” in which events can be built up inside a set of generators. This allows for the modularisation of event generators—wherein different event generators could be employed for different steps or components of the event generation process (illustrated in Figure 1).²



Figure 1: HepMC supports the concept of modularised event generation (illustrated above) by containing sufficient information within the event record to act as a messenger between two modular steps in the event generation process.

Physics events are generally visualised using diagrams with a graph structure (Figure 2, left) which HepMC imitates by separating out particles from vertices and representing them as the edges and nodes respectively of a graph³ (Figure 2, right). Each vertex maintains a listing of its incoming and outgoing particles, while each particle points back to its production vertex and decay vertex. The extension to multiple collisions is natural - the super-position of graphs from several different initial processes - and so the event may contain an unlimited number of (possibly interconnected) graphs. The number of vertices/particles in each event is also open-ended. A subset of the event (such as one connected graph or a single vertex and its descendants) may be examined or modified

¹<http://lcgapp.cern.ch/project/simu/HepMC/>

²At the *Physics at TeV Colliders Workshop 2001* in Les Houches, France, a group of Monte Carlo authors and experimentalists produced a document [4] which outlines the information content necessary for two event generators to communicate information about a hard process to the subsequent stages of event generation. This was implemented in a set of Fortran common blocks, and many ideas from HepMC were used, such as the scheme for handling color flow information. Version 1.1 of HepMC supports the full event information content of Ref. [4] (run information—pertaining to a collection of events—is also specified in that document and is not addressed in HepMC).

³Ref. [5] uses a similar structure.

without having to interpret complex parent/child relationship codes or re-shuffle the rest of the event record.

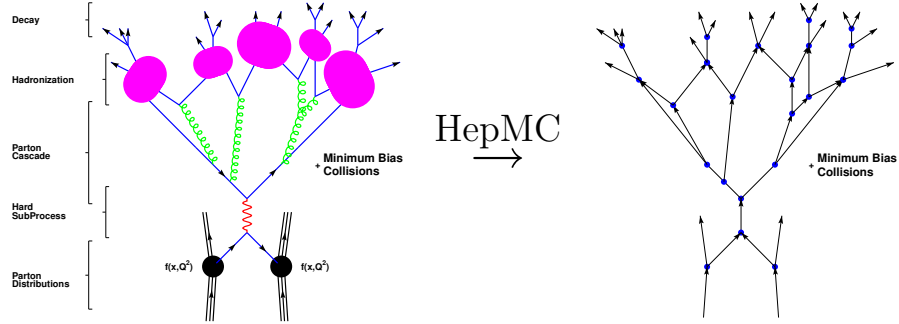


Figure 2: Events in HepMC are stored in a graph structure (right), similar to a physicist's visualization of a collision event (left).

1.1 Features of the HepMC Event Record

- simple - easy access to information provided by iterators
- minimum dependencies
- information is stored in a graph structure, physically similar to a collision event
- allows for the inclusion of spin density matrices
- allows for the tracing of an arbitrary number of flow patterns
- ability to store the state of random number generators (as integers)
- ability to store an arbitrary number of event weights
- strategies for conversion to/from HEPEVT?? which are easily extendible to support other event records
- strategies for input/output to/from ascii files which are easily extendible to support other forms of persistency

2 HepMC 2

Since January 2006, HepMC has been supported as an LCG external package. The official web site is now <http://savannah.cern.ch/projects/hepmc/>, and compiled libraries for supported platforms are available at [/afs/cern.ch/sw/lcg/external/HepMC](http://afs.cern.ch/sw/lcg/external/HepMC).

Historically, HepMC has used CLHEP (Ref. [6]) Lorentz vectors. Some users wished to use a more modern Lorentz vector package. At the same time, there was concern about allowing dependencies on any external package. Therefore, the decision was made to replace the CLHEP Lorentz vectors with a minimal vector representation within HepMC.

Because this is a major change, the versioning is changed from 1.xx.yy to 2.xx.yy. Normally, a version number change in *xx* represents a change to the code and a version number change in *yy* represents a bug fix.

There have also been continuing requests for other features. Changes to HepMC must be approved by the LCG simulation project.

2.1 Overview of Changes Since HepMC 1.26

See the HepMC ChangeLog [7] for a complete listing.

GenEvent now contains pointers to a heavy ion class and a PDF information class. The pointers are null by default.

GenParticle momentum and GenVertex position are represented by a simple FourVector class instead of the CLHEP Lorentz vectors. The SimpleVector.h header contains the FourVector and ThreeVector classes. GenVertex will return the ThreeVector portion of the position. Polarization will accept or return a ThreeVector representation of the polarization.

Both FourVector and ThreeVector have templated constructors. These constructors allow you to use the GenParticle and GenVertex constructors with *any* Lorentz vector, as long as the Lorentz vector has `x()`, `y()`, `z()`, and `t()` methods.

The generated mass, which has always been part of the HEPEVT common block, is now stored in GenParticle. When a particle has large momentum and small mass, calculating the mass from the momentum is unreliable. Also, different machine representations and roundoff errors mean that a calculated mass is not always consistent. If no generated mass is set, then the mass is calculated from the momentum and stored in GenParticle.

The IO_AsciiParticles class provides output in the Pythia style. This output is intended for ease of reading event output, not for persistency.

The IO_Ascii output class is deprecated in favor of IO_GenEvent. IO_GenEvent persists all information in the updated GenEvent object and uses iostreams for greater flexibility. IO_GenEvent also has a constructor taking a file name and mode type for backwards compatibility. Output remains in ascii format.

3 Package Structure

Entries within the event record are separated into particles and vertices. Each particle is composed of momentum, flow, and polarization classes as well as id and status information. The vertices are the connection nodes between particles and are a container class for particles: thus each particle within an event belongs to at least one vertex. In addition the vertex is composed of position, id, and (spin density matrix) weight information. Particles and vertices are uniquely identified by an integer—referred to as a “barcode”—which is meant to be a persistent label for a particular particle instance. The event is the container class for all (possibly inter-connected) vertices in the event and contains process id, event number, weight, and random number state information.

Iterators are provided as methods of the vertex and event classes which allow easy directed access to information in the C++ Standard Template Library (STL) style.

The event record class relationships and the particle data class relationships are shown in Figure 3.

Several input/output strategies are provided. The interface for these strategies is defined by an abstract base class, IO_BaseClass. These strategies are capable of input/output of events and as such they depend directly on the event record class.

The package consists of about 5500 lines of code including ≈ 1500 comments. There are 7 core classes (GenEvent, GenVertex, GenParticle, Flow, Polarization, WeightContainer, IO_BaseClass)

and several utility classes (i.e. `IO_HEPEVT`, `IO_GenEvent`, `HEPEVT_Wrapper`, `PythiaWrapper`, ...).

3.1 Dependencies

The HepMC 2 package depends only on the C++ Standard Template Library [8] (STL).

The HepMC 1 package dependencies were limited to STL and the vector classes from the Class Library for High Energy Physics [6] (CLHEP). Simple wrappers for the Fortran versions of Pythia [9] and Herwig are supplied with the package to allow the inclusion of event generation examples.

3.2 Namespace

The HepMC package is written within the `HepMC::` namespace. Outside of this namespace all class names will need to be prefixed by `HepMC::`.

3.3 Performance

The CPU time performance of the HepMC event record has been quantified by generating 1000 LHC $W\gamma$ production events using Pythia 5.7 and transferring the event record to HepMC using the `IO_HEPEVT` strategy. Results are summarised in Table 1. Generation of events in Pythia required 29 seconds of CPU time. Generating the same events and transferring them into HepMC required 34 seconds.

	CPU Time	File Size
Pythia	29 sec	
+ HepMC	34 sec	
+ HepMC + IO_Ascii	64 sec	60.5 Mbytes
+ LCWRITE	92 sec	106 Mbytes
HEPEVT raw size 1K events, 500K particles		48.2 Mbytes

Table 1: CPU time performance and file size using a dedicated 450 MHz Pentium III.

The time to write HepMC events as ascii files using the `IO_Ascii` strategy was compared to `LCWRITE`, a simple Fortran routine used in NLC studies to write the HEPEVT common block in formatted ascii to file. Generating the events with Pythia, transferring them to HepMC, and writing them to file took 64 seconds and produced a 60.5 Mbyte file. Generating the events with Pythia and writing them to file using the `LCWRITE` subroutine took 92 seconds and produced a 106 Mbyte file. Compression algorithms (such as `gzip`) can reduce the file sizes by a factor 3 or more. The raw size of the HEPEVT common block for these 1000 events (which in this case produced about 500K particles) is 48.2 Mbytes. In both cases most CPU time is spent writing to file. HepMC benefits from added logic when interpreting the record and from position information which is stored once for each vertex, rather than with every particle.

CPU time savings will be realized when HepMC is used inside event generators - since it is possible to target and modify one area of the particle/vertex graph without re-shuffling the rest of the event record.

4 Overview of Core Classes

NOTE ABOUT UNITS: HepMC does not define which units are used for the information stored in the event record. The HEPEVT standard uses GeV/mm, and so the output from most Fortran generators will normally be in these units. CLHEP uses MeV/mm, and some collaborations such as ATLAS have adopted these units for their simulation. Due to this ambiguity all mention of units has been removed from the HepMC documentation. HepMC users should refer to the code that fills the event record to determine which units are being used.

4.1 HepMC::GenEvent

IMPORTANT PUBLIC METHODS

- **add_vertex:** *adopts the specified vertex to the event and assumes responsibility for deleting the vertex*
- **remove_vertex:** *removes the specified vertex from the event, the vertex is not deleted - thus use of this method is normally followed directly by a delete vertex operation*
- **vertex_iterator:** *iterates over all vertices in the event - described in the iterator section*
- **particle_iterator:** *iterates over all particles in the event - described in the iterator section*
- **vertex_const_iterator:** *constant version of the vertex_iterator*
- **particle_const_iterator:** *constant version of the particle_iterator*
- **print:** *gives a formatted printout of the event to the specified output stream*
- **barcode_to_particle:** *returns a pointer to the particle associated with the integer barcode argument*
- **barcode_to_vertex:** *returns a pointer to the vertex associated with the integer barcode argument*

RELEVANT DATA MEMBERS

- **signal_process_id:** *an integer ID uniquely specifying the signal process (i.e. MSUB in Pythia).*
- **event_number:** *integer*
- **event_scale:** *(optional) the scale of this event. (-1 denotes unspecified)*
- **alphaQCD:** *(optional) the value of the strong coupling constant α_{QCD} used for this event. (-1 denotes unspecified)*
- **alphaQED:** *(optional) the value of the electroweak coupling constant α_{QED} (e.g. $\frac{1}{128}$) used for this event. (-1 denotes unspecified)*
- **signal_process_vertex:** *(optional) pointer to the vertex defined as the signal process - allows fast navigation to the core of the event*
- **beam_particle_1:** *(optional) pointer to the first incoming beam particle*
- **beam_particle_2:** *(optional) pointer to the second incoming beam particle*
- **weights:** *a container of an arbitrary number of 8 byte floating point event weights*
- **random_states:** *a container of an arbitrary number of 4 byte integers which define the random number generator state just before the event generation*
- **heavy_ion:** *(optional) a pointer to a HeavyIon object (zero by default)*
- **pdf_info:** *(optional) a pointer to a PdfInfo object (zero by default)*

NOTES AND CONVENTIONS

- if hit and miss Monte Carlo integration is to be performed with a single event weight, the first weight will be used by default
- Memory allocation: vertex and particle objects will normally be created by the user with the NEW operator. Once a vertex (particle) is added to a event (vertex), it is "adopted" and becomes the responsibility of the event (vertex) to delete that vertex (particle).

The GenEvent is the container class for vertices. A listing of all vertices is maintained with the event, giving fast access to vertex information. GenParticles are accessed by means of the vertices.

Extended event features (weights, random_states, heavy_ion, pdf_info) have been implemented such that if left empty/unused performance and memory usage will be similar to that of an event without these features.

Iterators are provided as members of the GenEvent class (described in Section 5). Methods which fill containers of particles or vertices are *not provided*, as the STL provides these functionalities

with algorithms such as `copy` and iterator adaptors such as `back_inserter` giving a clean generic approach. Using this functionality it is easy to obtain lists of particles/vertices given some criteria - such as a list of all final state particles. Classes which provide the criteria (called predicates) are also *not provided*, as the number of possibilities is open ended and specific to the application - and would clutter the HepMC package. Implementing a predicate is simple (about 4 lines of code). Examples are given in the GenEvent header file and in `example_UsingIterators.cc` (Section 7).

The `signal_process_id` is packaged with each event (rather than being associated with a run class for example) to handle the possibility of many processes being generated within the same run. A container of tags specifying the meaning of the weights and `random_states` entries is envisioned as part of a run class - which is beyond the scope of an event record.

4.1.1 HepMC::PdfInfo

RELEVANT DATA MEMBERS

- **id1**: *flavour code of first parton*
- **id2**: *flavour code of second parton*
- **x1**: *fraction of beam momentum carried by first parton ("beam side")*
- **x2**: *fraction of beam momentum carried by second parton ("target side")*
- **scalePDF**: *Q-scale used in evaluation of PDF's (in GeV)*
- **pdf1**: *PDF (id1, x1, Q)*
- **pdf2**: *PDF (id2, x2, Q)*

PdfInfo stores additional PDF information for a GenEvent. Creation and use of this information is optional.

4.1.2 HepMC::HeavyIon

RELEVANT DATA MEMBERS

- **Ncoll_hard**: *Number of hard scatterings*
- **Npart_proj**: *Number of projectile participants*
- **Npart_targ**: *Number of target participants*
- **Ncoll**: *Number of NN (nucleon-nucleon) collisions*
- **N_Nwounded_collisions**: *Number of N-Nwounded collisions*
- **Nwounded_N_collisions**: *Number of Nwounded-N collisions*
- **Nwounded_Nwounded_collisions**: *Number of Nwounded-Nwounded collisions*
- **spectator_neutrons**: *Number of spectators neutrons*
- **spectator_protons**: *Number of spectators protons*
- **impact_parameter**: *Impact Parameter(fm) of collision*
- **event_plane_angle**: *Azimuthal angle of event plane*
- **eccentricity**: *eccentricity of participating nucleons in the transverse plane (as in phobos nucl-ex/0510031)*
- **sigma_inel_NN**: *nucleon-nucleon inelastic (including diffractive) cross-section*

HeavyIon provides additional information storage in GenEvent for Heavy Ion generators. Creation and use of this information is optional.

4.2 HepMC::GenVertex

IMPORTANT PUBLIC METHODS

- **add_particle_in**: *adds the specified particle to the container of incoming particles*
- **add_particle_out**: *adds the specified particle to the container of outgoing particles*
- **remove_particle**: *removes the specified particle from both/either of the incoming/outgoing particle containers, the particle is not deleted - thus use of this method is normally followed directly by a delete particle operation*
- **vertex_iterator**: *iterates over vertices in the graph, given a specified range - described in the iterator section*

- **particle_iterator**: iterates over particles in the graph, given a specified range - described in the iterator section

RELEVANT DATA MEMBERS

- **position**: \vec{x}, ct stored as *FourVector*
- **id**: integer id, may be used to specify a vertex type
- **weights**: a container of 8 byte floating point numbers of arbitrary length, could be mapped in pairs into rows and columns to form spin density matrices of complex numbers
- **barcode**: an integer which uniquely identifies the *GenVertex* within the event. For vertices the barcodes are always negative integers.

NOTES AND CONVENTIONS

- no standards are currently defined for the vertex id
- once a particle is added, the vertex becomes its owner and is responsible for deleting the particle

The *GenVertex* is the container class for particles and forms the nodes which link particles into a graph structure.

The weights container is included with each vertex with the intention of storing spin density matrices. It is envisioned that a generator package would assign spin density matrices to particle production vertices and provide the functional form of the frame definition for the matrix as a “look-up” method for interpreting the weights. The generator package would also provide a boost method to go from the frame of the density matrix to the lab frame and back without destroying correlations. This gives maximum freedom to the sub-generators - allowing for different form definitions. This implementation is consistent with the EvtGen B-decay package [10] requirements.

4.3 HepMC::WeightContainer

RELEVANT DATA MEMBERS

- **weights**: a vector of 8-byte floating point weights

NOTES AND CONVENTIONS

- methods are coded and names chosen in the spirit of the STL vector class

The *WeightContainer* is just a storage area for double precision weights used in *GenEvent* and *GenVertex*. It is essentially an interface to the STL vector class, and its member functions are chosen in that spirit. You might, for instance, use the *GenEvent* weights to include information about differential cross sections.

4.4 HepMC::GenParticle

IMPORTANT PUBLIC METHODS

- **operator FourVector**: conversion operator - resolves the particle as a 4-vector according to its momentum
- **generatedMass**: generated mass
- **momentum().m()**: calculates mass from momentum

DATA MEMBERS

- **momentum**: \vec{p}, cE stored as *FourVector*
- **generated_mass**: generated mass for this particle
- **pdg_id**: unique integer ID specifying the particle type
- **status**: integer specifying the particle’s status (i.e. decayed or not)
- **flow**: allows for the storage of flow patterns (i.e. color flow), refer to *Flow* class
- **polarization**: stores the particle’s polarization as (θ, ϕ) , refer to *Polarization* class
- **production_vertex**: pointer to the vertex where the particle was produced, can only be set by the vertex
- **end_vertex**: pointer to the vertex where the particle decays, can only be set by the vertex
- **barcode**: an integer which uniquely identifies the *GenParticle* within the event. For particles the barcodes are always positive integers.

NOTES AND CONVENTIONS

- the particle ID should be specified according to the PDG standard [11]
- status codes are as defined for HEPEVT [12]⁴

The particle is the basic unit within the event record. The `GenParticle` class is composed of the `FourVector`, `Flow`, and `Polarization` classes.

Pointers to the particle’s production and end vertex are included. In order to ensure consistency between vertices/particles - these pointers can only be set from the vertex. Thus adding a particle to the `particles.in` container of a vertex will automatically set the `end_vertex` of the particle to point to that vertex.

The definition of a `HepLorentzVector` scope resolution operator allows for the use of 4-vector algebra with particles (i.e. preceding an instance, `particle`, of the `HepMC::GenParticle` class by `(HepLorentzVector)particle` causes it to behave exactly like its 4-vector momentum, examples are given in the particle header file).

A second 4-vector for the particle’s momentum at decay time has *not* been included (as for example in [5], where the second momentum vector is included to facilitate tracking through material). If this is desirable, one can simply add a decay vertex with the same particle type going out. This is intuitive, since a change in momentum cannot occur without an interaction (vertex).

4.4.1 HepMC::Flow

IMPORTANT PUBLIC METHODS

- **`connected_partners`**: *returns a container of all particles connected via the specified flow pattern*
- **`dangling_connected_partners`**: *returns a container of all particles “dangling” from the ends of the specified flow pattern*

RELEVANT DATA MEMBERS

- **`particle_owner`**: *points back to the particle to which the flow object belongs*
- **`icode map`**: *container of integer flow codes - each entry has an index and an icode*

NOTES AND CONVENTIONS

- code indices 1 and 2 are reserved for color flow

The `Flow` class is a data member of the `GenParticle`—its use is optional. It stores flow pattern information as a series of integer flow codes and indices. This method features the possibility of storing non-conserved flow patterns (such as baryon number violation in SUSY models). Some examples of integer flow code representation for several events are provided in Ref. [4].

The `Flow` class is used to keep track of flow patterns within a graph - each pattern is assigned a unique integer code, and this code is attached to each particle through which it passes. Different flow types are assigned different flow indices, i.e. color flow uses index 1 and 2. Methods are provided to return a particle’s flow partners. An example is given at the top of the `Flow` header file.

⁴For convenience the HEPEVT standard status codes are enumerated:

0	null entry
1	existing entry - not decayed or fragmented, represents the final state as given by the generator
2	decayed or fragmented entry
3	documentation line
4-10	undefined, reserved for future standards
11-200	at the disposal of each model builder - equivalent to a null line
201-	at the disposal of the user, in particular for event tracking in the detector

4.4.2 HepMC::Polarization

RELEVANT DATA MEMBERS

- **theta:** θ angle in radians $0 \leq \theta \leq \pi$
- **phi:** ϕ angle in radians $0 \leq \phi < 2\pi$

NOTES AND CONVENTIONS

- the angles are robust - if you supply an angle outside the range, it is properly translated (i.e. 4π becomes 0)

Polarization is a data member of GenParticle - its use is optional. It stores the (θ, ϕ) polarization information which can be returned as a ThreeVector as well.

4.4.3 HepMC::FourVector

IMPORTANT PUBLIC METHODS

- A number of simple vector manipulations are available. Check the reference manual for details.

RELEVANT DATA MEMBERS

- **x:** position x or momentum px
- **y:** position y or momentum py
- **z:** position z or momentum pz
- **t:** time or energy

GenParticle momentum and GenVertex position are stored as FourVectors. FourVector has a templated constructor that will automatically convert any other vector with $x()$, $y()$, $z()$, and $t()$ access methods to a FourVector. This feature is used when converting from the HEPEVT common block.

4.5 HepMC::IO_BaseClass

IMPORTANT PUBLIC METHODS

- **write_event:** writes out the specified event to the output strategy
- **read_next_event:** reads the next event from the input strategy into memory
- **write_particle_data_table:** writes out the specified particle data table to the output strategy
- **read_particle_data_table:** reads a particle data table from the input strategy
- **operator<<,operator>>:** overloaded to give the same results as any of the above methods

IO_BaseClass is the abstract base class defining the interface and syntax for input and output strategies of events and particle data tables.

Several IO strategies are supplied:

- **IO_GenEvent** uses iostreams for input and output thereby providing a form of persistency for the event record. This class handles all information found in a GenEvent object. This class replaces IO_Ascii and reads both formats. Events may be contained within the same file together with an unlimited number of comments. The examples (Section 7) make use of this class.
- **IO_Ascii** is deprecated.
- **IO_AsciiParticles** writes events in a format similar to Pythia 6 output. This is intended for human readability.
- **IO_HEPEVT** reads and writes events to/from the Fortran HEPEVT common block. It relies on a helper class HEPEVT_Wrapper which is the interface to the common block (which is

defined in the header file `HEPEVT_Wrapper.h`⁵). This IO strategy provides the means for interfacing to Fortran event generators. Other strategies which interface directly to the specific event record of a generator could be easily implemented in this style. An example of using `IO_HEPEVT` to transfer events from Pythia into HepMC is given in `example_MyPythia.cc` (Section 7).

5 Overview of Iterators

Examples of using the particle/vertex iterators are provided in `example_UsingIterators.cc` (Section 7).

5.1 HepMC::GenEvent::vertex_iterator

`GenEvent::vertex_iterator` inherits from `std::iterator<std::forward_iterator_tag,...>`. It walks through all vertices in the event exactly once. It is robust and fast, and provides the best way to loop over all vertices in the event. For each event, `vertices_begin()` and `vertices_end()` define the beginning and one-past-the-end of the particle iterator respectively.

5.2 HepMC::GenEvent::vertex_const_iterator

A constant version of `HepMC::GenEvent::vertex_iterator`, otherwise identical.

5.3 HepMC::GenEvent::particle_iterator

`GenEvent::particle_iterator` inherits from `std::iterator<std::forward_iterator_tag,...>`. It walks through all particles in the event exactly once. It is robust and fast, and provides the best way to loop over all particles in the event. For each event, `particles_begin()` and `particles_end()` define the beginning and one-past-the-end of the particle iterator respectively.

5.4 HepMC::GenEvent::particle_const_iterator

A constant version of `HepMC::GenEvent::particle_iterator`, otherwise identical.

5.5 HepMC::GenVertex::vertex_iterator

NOTES AND CONVENTIONS

- the iterator range must be specified to instantiate - choices are: parents, children, family, ancestors, descendants, and relatives
- note: iterating over all ancestors and all descendents is *not* necessarily equivalent to all relatives - this is consistent with the range definitions

⁵Different conventions exist for the fortran HEPEVT common block. 4 or 8-byte floating point numbers may be used, and the number of entries is often taken as 2000 or 4000. To account for all possibilities the precision (float or double) and number of entries can be set for the wrapper at run time,

i.e. `HEPEVT_Wrapper::set_max_number_entries(4000);`
`HEPEVT_Wrapper::set_sizeof_real(8);` .

To interface properly to HEPEVT and avoid nonsensical results, it is essential to get these definitions right *for your application*. See `example_MyPythia.cc` (Section 7) for an example.

GenVertex::vertex_iterator differs from GenEvent::vertex_iterator in that it has both a starting point and a range. The starting point is the vertex - called the root - from which the iterator was instantiated, and the range is defined relative to this point. The possible ranges are defined by an enumeration called HepMC::IteratorRange and the possibilities are:

- **parents:** *walks over all vertices connected to the root via incoming particles*
- **children:** *walks over all vertices connected to the root via outgoing particles*
- **family:** *walks over all vertices connected to the root via incoming or outgoing particles*
- **ancestors:** *walks over all vertices connected to the root via any number of incoming particle edges - i.e. returns the parents, grandparents, great-grandparents, ...*
- **descendants:** *walks over all vertices connected to the root via any number of outgoing particle edges - i.e. returns the children, grandchildren, great-grandchildren, ...*
- **relatives:** *walks over all vertices belonging to the same particle/vertex graph structure as the root*

The iterator algorithm traverses the graph by converting it to a tree (by “chopping” the edges at the point where a closed cycle connects to an already visited vertex) and returning the vertices in post order. The iterator requires more logic than the GenEvent::vertex_iterator and thus access time is slower (the required to return one vertex goes like $\log n$ where n is the number of vertices already returned by the iterator).

GenVertex::vertex_iterator allows the user to step into a specific part of a particle/vertex graph and obtain targetted information about it.

5.6 HepMC::GenVertex::particle_iterator

NOTES AND CONVENTIONS

- the iterator range must be specified to instantiate - choices are: parents, children, family, ancestors, descendants, and relatives

GenVertex::particle_iterator behaves exactly like GenVertex::vertex_iterator, with the exception that it returns particles rather than vertices. As a particle defines an edge or line (rather than a point) in the particle/vertex graph, it is intuitive to define the particle_iterator relative to a vertex (point in the graph) - thus the starting point (root) is still a vertex, and the range is defined relative to this root. The extension to particles can be made by using the particle’s production or end vertex as the root. Possible ranges are defined by an enumeration called HepMC::IteratorRange and the possibilities are:

- **parents:** *walks over all particles incoming to the root*
- **children:** *walks over all particles outgoing from the root*
- **family:** *walks over all particles incoming or outgoing from the root*
- **ancestors:** *walks over all incoming particles or particles incoming to ancestor vertices of the root - i.e. returns the parents, grandparents, great-grandparents, ...*
- **descendants:** *walks over all outgoing particles or particles outgoing to descendant vertices of the root - i.e. returns the children, grandchildren, great-grandchildren, ...*
- **relatives:** *walks over all particles belonging to the same particle/vertex graph structure as the root*

The class is composed of a GenVertex::vertex_iterator - and the same considerations apply.

6 Building HepMC

Source code, binary and source code tarballs, bug tracking, etc. are all available from the HepMC web pages [13] at <https://savannah.cern.ch/projects/hepmc/>.

Source code tarballs are on the download page: <http://lcgapp.cern.ch/project/simu/HepMC/download/>.

THE FOLLOWING RECIPE IS A GUIDELINE AND SHOULD BE MODIFIED ACCORDING TO TASTE.

- **download source code tarball:**
- **mkdir cleanDIR:** *Make a new directory to work in.*
- **cd cleanDIR:**
- **tar -xzf HepMCTarball:** *Unwind the tarball you downloaded.*
- **mkdir build install:** *Define directories for building and installation.*
- **cd build:** *This is your real working directory.*
- **../HepMC-release/configure --prefix=../install:** *--prefix tells the tools where to install the library and headers. The default install location is /usr/local.*
- **make:** *Compile HepMC.*
- **make check:** *Run the tests. This is optional but strongly recommended.*
- **make install:** *Install everything in your specified directory.*

Binary downloads are available for some releases.

7 Examples

Examples are provided in the examples directory of the package and are installed in the installation directory under examples/HepMC. Tests, found in the test directory of the package, also provide useful examples. The tests are not installed.

- **Using the HepMC vertex and particle iterators:** `example_UsingIterators.cc`
- **Using HepMC with Pythia (Fortran):** `example_MyPythia.cc`, `example_MyPythiaOnlyToHepMC.cc`, and `example_PythiaParticle.cc`
- **An Event Selection with Pythia Events:** `example_MyPythiaWithEventSelection.cc`
- **Event Selection and Ascii IO** `example_EventSelection.cc`
- **Using HepMC with Herwig:** `example_MyHerwig.cc`
- **Write an event file and then read it:** `example_MyPythiaRead.cc`
- **Building an Event from Scratch in the HepMC Framework:** `example_BuildEventFromScratch.cc`
- **Verify that copying generated events behaves as expected:** `testHerwigCopies.cc` and `testPythiaCopies.cc`

All examples use `IO_GenEvent` instead of the deprecated `IO_Ascii`.

8 Deprecated Classes

Two major classes have been deprecated: `IO_Ascii` and `ParticleData`. `IO_Ascii` is replaced by `IO_GenEvent`, which uses iostreams instead of files.

The `ParticleData` classes had become outmoded and would need a lot of work. Instead, we recommend using packages already developed for this purpose, such as HepPDT [14].

8.1 HepMC::IO_Ascii (deprecated)

NOTES AND CONVENTIONS

- **IO_Ascii** reads and writes events and particle data tables to files in machine readable ascii, thereby providing a form of persistency for the event record. Events and particle data tables may be contained within the same file (recommend to write the particle data table first to save access time) together with an unlimited number of comments. IO_GenEvent will read files written by IO_Ascii.

8.2 HepMC::ParticleData (deprecated)

RELEVANT DATA MEMBERS

- **name:** *std::string* giving an ascii description of the particle type
- **pdg_id:** unique ID integer denoting the particle type as defined by the PDG
- **charge:** in fraction of proton charge
- **mass:** in energy units
- **c×lifetime:** particle lifetime in [mm]
- **spin:**

NOTES AND CONVENTIONS

- the lifetime can be set by specifying either the lifetime or the width
- c×lifetime=-1 specifies a stable particle (zero width)

Data for each particle type (mass, lifetime, charge, etc.) can be stored as particle data objects for which a particle data table container is provided. There are no dependencies between the particle data objects and the other elements of the event - the relationship exists only by means of the particle id which is used to lookup information from within a particle data table. As such the data table and event record are separate modular entities which need not be used in conjunction (a user may choose to employ the event record while using his own particle data classes).

ParticleData class stores information about a particular particle type. It is intended that a different ParticleData object is created for each particle and each anti-particle - necessary for CP violation cases. The charge and spin are stored internally as integers representing $\frac{\text{proton charge}}{3}$ and $\frac{\text{photon spin}}{2}$ respectively.

8.3 HepMC::ParticleDataTable (deprecated)

IMPORTANT PUBLIC METHODS

- **find:** returns the ParticleData instance with the specified pdg_id
- **operator[]:** equivalent to find
- **insert:** includes the specified ParticleData instance in the table
- **erase:** removes the specified ParticleData instance from the table but does not delete it
- **iterator/const_iterator:** iterates over all entries in the table
- **make_antiparticles_from_particles:** for each charged entry in the table, makes an equivalent entry with opposite charge and pdg_id.
- **delete_all:** removes all ParticleData instances from the table and deletes them
- **merge_table:** merges the entries from the specified ParticleDataTable if they are not already in the current table
- **print:** gives a formatted printout of the table to the specified output stream

RELEVANT DATA MEMBERS

- **description:** *ascii* description of the table stored as *std::string*
- **data_table:** container of pointers to ParticleData instances mapped onto their associated pdg_id's

ParticleDataTable is a container for ParticleData instances - it is basically just an interface to an STL map, and STL naming conventions are employed. A ParticleData instance may belong to any number of ParticleDataTables. The ParticleDataTable is not the owner of the ParticleData instances and is not responsible for deleting them (though a delete_all method is provided). Two ParticleData instances with identical pdg_id's are forbidden from entering the same ParticleDataTable.

9 Acknowledgements

We would like to acknowledge useful suggestions, consultations, and comments from: Ian Hinchliffe, Pere Mato, H.T. Phillips, Anders Ryd, Maria Smizanska, and Brian Webber. R.D. Schaffer and Lassi Tuura provided many useful suggestions on the package architecture. Thanks to Witold Pokorski and Pere Mato for providing the fixes that make HepMC compile and run on Windows with Microsoft Visual C++.

References

- [1] M. Dobbs and J.B. Hansen, “The HepMC C++ Monte Carlo Event Record for High Energy Physics”, Computer Physics Communications (to be published) [ATL-SOFT-2000-001].
- [2] Pythia 8.1 available at <http://www.thep.lu.se/torbjorn/pythiaaux/present.html>.
- [3] Herwig++ 2.1 available at <http://projects.hepforge.org/herwig/>.
- [4] E. Boos *et al.*, “Generic user process interface for event generators,” arXiv:hep-ph/0109068.
- [5] S. Protopopescu, “MC++ Interface”. Available from <http://ox3.phy.bnl.gov/~serban/mcpp/index.html>.
- [6] “A Class Library for High Energy Physics,” (CLHEP). Available from <http://wwwasd.web.cern.ch/wwwasd/lhc++/clhep/>.
- [7] Latest HepMC ChangeLog available at <http://simu.cvs.cern.ch/cgi-bin/simu.cgi/simu/HepMC/ChangeLog?view=markup>.
- [8] A.A. Stepanov, M. Lee, “The Standard Template Library,” Hewlett-Packard Laboratories Technical Report HPL-94-34, April 1994, revised July 7, 1995. Available from <ftp://butler.hpl.hp.com/stl/>.
- [9] T. Sjostrand *et al.*, “High-energy physics event generation with PYTHIA 6.1,” Comput. Phys. Commun. **135**, 238 (2001).
- [10] A. Ryd, D. Lange, “The EvtGen package for simulating particle decays,” Computing in High Energy Physics, Chicago, Illinois, USA (1998).
- [11] W.-M. Yao *et al.*, “Review of particle physics,” Journal of Physics **G33**, 1 (2006). Available from <http://pdg.lbl.gov/>.
- [12] L. Garren, “StdHep 5.05 Monte Carlo Standardization at FNAL,” Fermilab PM0091. Available from <http://cepa.fnal.gov/psm/stdhep/>.
- [13] “a C++ Event Record for Monte Carlo Generators,” (HepMC). Available from <https://savannah.cern.ch/projects/hepmc/>.
- [14] HepPDT is available at <http://savannah.cern.ch/projects/heppdt/>.

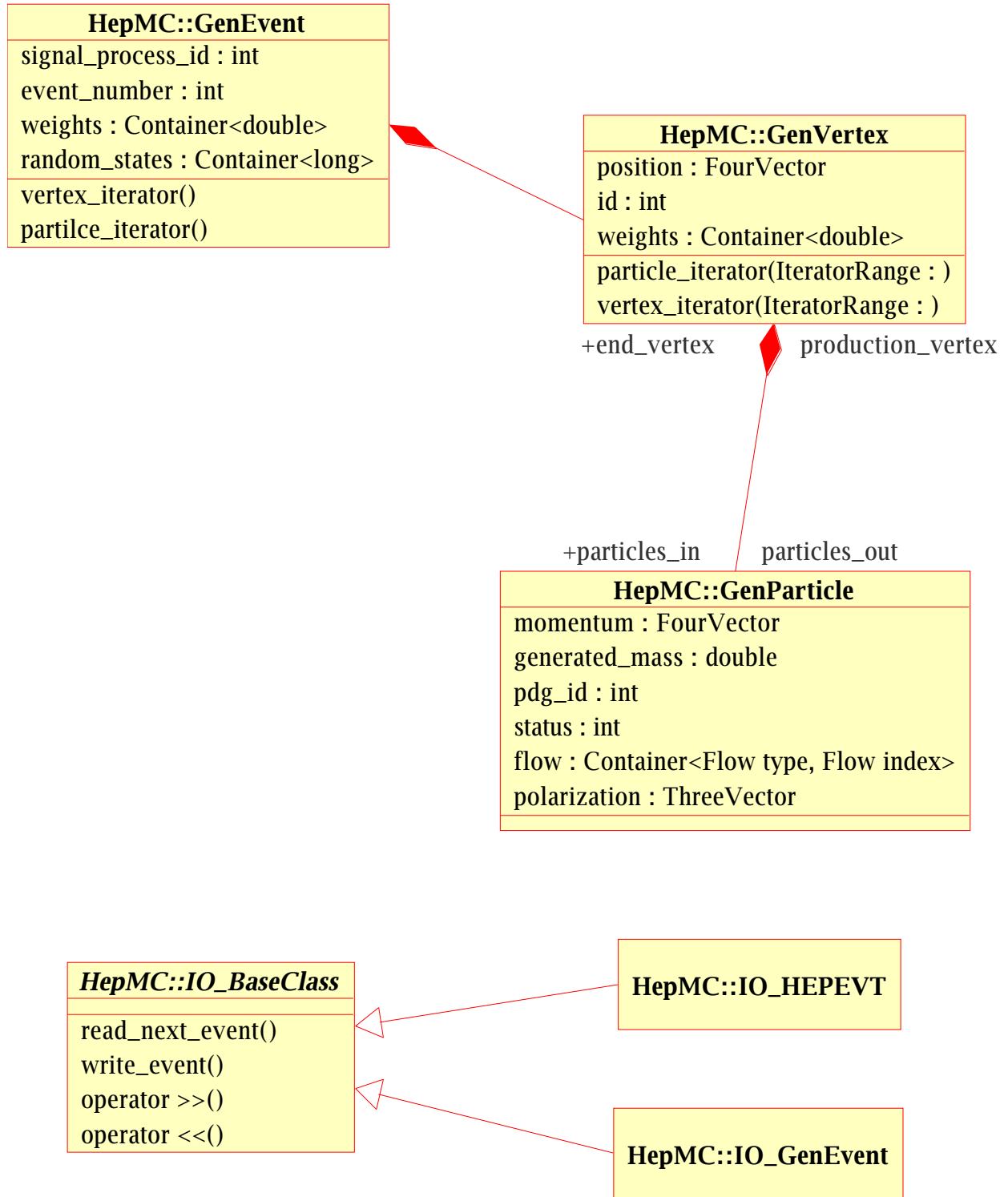


Figure 3: Class diagrams for the event record classes (GenEvent, GenVertex, and GenParticle) and the IO strategies are shown in the UML notation.