Computer Physics Communications 71 (1992) 1–14
North-Holland

Computer Physics
Communications

The MC++ event generator toolkit – version 0

Leif Lönnblad [1]

Deutsches Elektronen Synchrotron - DESY, Notkestraße 85, W-2000 Hamburg 52, Germany

and

Anders Nilsson [2]

Department of Theoretical Physics, University of Lund, Sölvegatan 14A, S-22362 Lund, Sweden

Received 15 January 1992

We present a toolkit, written in the C++ programming language, for event generation in high energy physics. The toolkit, called MC++, is an attempt to formulate the event generation chain in high energy particle collisions in a transparent and generic way using object oriented programming techniques.

1. Introduction

Monte Carlo Event Generators (MCEGs) have become invaluable tools in todays high energy physics experiments. They are used to "translate" theoretical predictions into experimental observables and to make reasonable estimations of backgrounds and systematic errors to these.

There are a number of these MCEGs "on the market" today. Some of them are complete in the sense that they simulate the complete event generating chain (e.g. HERWIG [1], ISAJET [2] and JETSET [3]) and some only simulate one or two parts and have to be interfaced to other programs to become complete (e.g. Ariadne [4] and HERACLES [5]).

The programs have some things in common. They all rely on the assumption that the event generating process can be treated more or less like a Markov chain, e.g. that the decay of a particle is independent of the way it is produced. Hence the event generation is divided into steps which are common for almost all programs. In e.g. a pp collision the interacting partons and the hard process are chosen from structure function parametrizations and according to the relevant matrix elements and a multi-partonic state is developed typically by using initial- and final-state parton showers. These partons are then hadronized using some phenomenological model and finally the produced hadrons are allowed to decay into stable particles.

The actual models for these different steps differ, however, very much between different programs, and the results produced also differ when extrapolating to future accelerators, even when they are tuned to fit the same set of available data [6].

The fact that today there exist e.g. several models for the hadronization which all reproduce present data to a satisfactory level is frequently used to estimate the fragmentation effects on theoretical predictions made on the partonic level. Ideally one would like to be able to

Correspondence to: L. Lönnblad, Deutsches Elektronen Synchrotron - DESY, Notkestraße 85, W-2000 Hamburg 52, Germany.

[1] E-mail: lonnblad@apollo3.desy.de (internet) and lonnblad@desyvax (bitnet).

[2] E-mail: anders@thep.lu.se (internet) and thepan@seldc52 (bitnet).

e.g. run a particular program which implements some model for the partonic state using, say, the Lund string fragmentation model [7] implemented in JETSET [3] and then do the same run only this time using the cluster fragmentation model [1] implemented in HERWIG. This is, however, easier said than done. Although all programs are written in FORTRAN, their internal interface between different modules are very different.

There has been some attempts to standardize the interface between different programs e.g. the HEPEVT common block described in ref. [8], but there is still a long way to go.

1.1. Object oriented programming

Recently a study was made [9] to see how object oriented programming (OOP) [10] can be used to facilitate both the interface between different MCEG programs and the actual development of new programs. As a consequence a project was started for developing a general toolkit for Monte Carlo event generating called MC++ written in C++ (see for example ref. [11]). In this paper we will describe a "zeroth" version of this toolkit.

Contrary to more conventional procedural languages, where subroutines are called to operate upon fundamental variables (which in MCEGs are typically stored in large arrays in common blocks), OOP is based on the concept of objects. These are generalized variables which communicate with each other by sending and receiving messages. (For an introduction to OOP we refer to reference [10].) This enables you to define classes of objects in close correspondence to the physical objects of interest. You can e.g. define a "particle" object which understands and can respond to messages like "What is your charge?", "Boost!" and "Decay!".

In this way we have written a number of object classes defining the event generating chain, thus in a way creating a "dialect" of the C++ language, or a toolkit, specially designed for MCEG.

The reason we chose C++ out of the handful of OOP languages available today was mainly a question of availability [#1]. Before settling for a "standard" OOP language for MCEG or indeed for HEP in general, a more thorough comparison of existing languages should of course be made. We, however, felt that it was important to first have an example as a basis for discussion, so we simply chose the most widely used language.

1.2. The goals for the MC++ project

The MC++ project was started with a number of goals in mind. Some of them were:

- The code should in a transparent way mirror the different parts of the event generating chain.
- The code should be modular, so that a model describing only a small part of the event generating chain is easily added.
- The code should provide utilities to facilitate the development of new "modules".
- The code should be easily interfaced to a graphical user interface.
- The user should easily be able to add, remove, and change decay channels, branching ration and decay methods etc. of particles as well as adjust their mass and lifetimes etc.
- The user should just as easily be able to change between different models for e.g. partonic showers and fragmentation.

The result was a structure where the whole event generating chain is defined in terms of (generalized) particles decaying into (generalized) particles. The particles are implemented as a class of objects in C++. More complex objects, such as a "Lund QCD string", are realized through the concept of inheritance and are "subclasses" of the generic particle class.

As it stands MC++ is not complete in any way. It can only handle, through the use of the "old" MCEG programs JETSET and Ariadne, e^+e^- collisions, but we still think it is advanced enough to serve as an example of what may be done with OOP for MCEG.

[#1] For example the C++ compiler called g++ written by the Free Software Foundation can be obtained free of charge via anonymous ftp to prep.ai.mit.edu

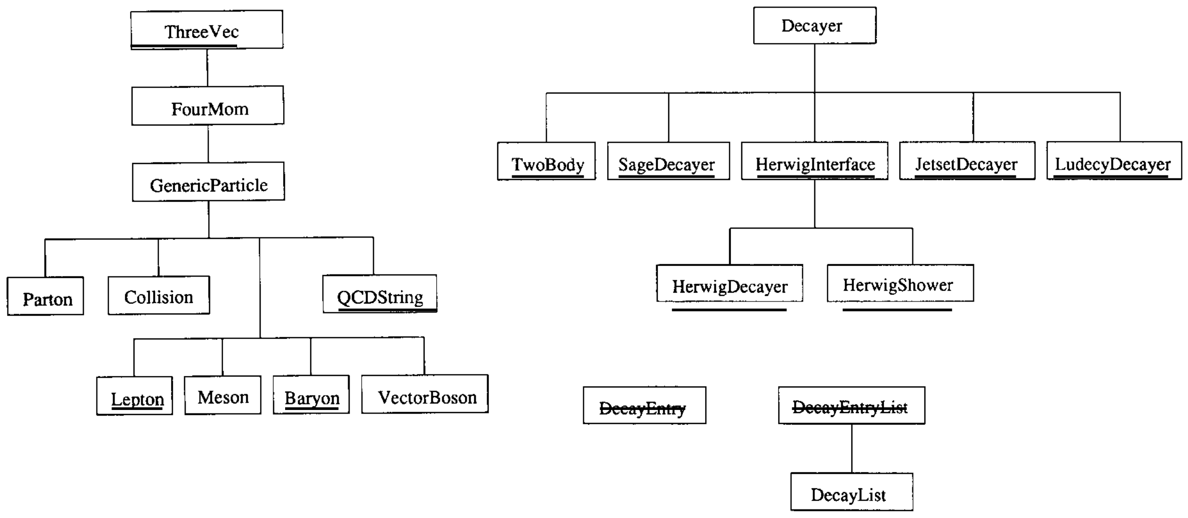

Fig. 1. The class hierarchy of MC++. Here we only show the most important classes of MC++. The toolkit contains a number of other classes not shown here.

The layout of the rest of this paper is as follows. In section 2 we first explain the class structure of MC++, describing the three fundamental classes "generic particle", "decayer" and "particle factory". Then in section 3 we explain how MC++ should be used, how to develop new "decayers" and how it may communicate with detector simulation and analysis programs. In the last section we discuss the future of MC++ and finally in the appendix information can be found on how to obtain and install MC++.

2. The structure of MC++

As mentioned in the introduction the event generating chain in MC++ is defined in terms of particles decaying into particles. This means that the concept of particles is somewhat broader than usual – also e.g. an e^+e^- collision is considered to be a particle, typically decaying into a $q\bar{q}$ or a $l\bar{l}$ pair. Most of this structure is embedded in the GenericParticle class.

2.1. The GenericParticle class

As seen in fig. 1 the GenericParticle class is a sub-class of FourMomentum, this means that a particle can be treated as just that, which is useful when developing decay models. The instance variables of the GenericParticle class can be divided into two groups, one which describes the generic features of a particle type – name, charge, mass, decay channels, etc. – and one describing a particular instance of a particle – fractional lifetime, helicity, a list of its children, etc. The decay channels are described as a pointer to a DecayList object, which is simply a list of objects of the DecayEntry class. Each DecayEntry contains a branching fraction, a pointer to a list of decay products and a pointer to a Decayer object (see below).

When a particle is told to decay, it will tell the DecayList to randomly select a DecayEntry according the branching fractions. Then it will tell the Decayer object pointed to by that DecayEntry to perform the decay, giving it pointers to itself and to the, possibly empty, list of decay

```
ParticleList &GenericParticle::Decay(){

   ParticleList *dummy = new ParticleList;

   if( IsStable() == false ){
      if( !hasDecayed ){
         DecayEntry* dec;
         if( decayTable == 0 ){
            FATAL("Has no decay table");
         }
         switch( decayTable→Length() ){
          case 0:
            return *dummy;
            break;
          case 1:
            dec = decayTable→First();
            if( dec→Decay(this) == 0 )
               return *dummy;
            break;
          default:
            float total = decayTable→GetTotal();
            float left = total;
            float tmp = (factory→rnd).Flat();
            dec = decayTable→SelectChannel(tmp);
            if( dec == 0 )
               return *dummy;
            while( dec→Decay(this) == 0 ){
               decayTable→Remove(dec);
               left-=dec→Branch();
               decayTable→Insert(dec);
               dec = decayTable→SelectChannel(
                     (factory→rnd).Flat(left/total));
               if( dec == 0 )
                  return *dummy;
            }
         }
         hasDecayed = YES;
      }
      return *childList;
   }
   else
      return *dummy;
}
```

Fig. 2. The Decay() method of the GenericParticle class, the method selects a decay entry according to the relevant branching ratio and then sends a message to the decay entry to decay itself. If this fails, the selection is redone, choosing one of the remaining channels.

products. Finally the particle will either tell its children to decay or return a list of its children to the caller which then explicitly can continue the decay chain. In this way the whole event generation is defined. The Decay() method for a GenericParticle is shown in fig. 2

The GenericParticle is a so-called abstract base class and should never actually be instantiated, instead all realizable particles belong to subclasses of the GenericParticle class. E.g. a π^0 is an object of the Meson class, which is a subclass of GenericParticle. In this way also very complicated objects can be treated on the same footing as particles, e.g. a Lund-type string consisting of a $q\bar{q}$ pair connected through a number of gluons can be treated in the same way as a π^0. As a matter of fact also the Decay() procedure is identical although the actual decay method applied is of course very different.

There are only a few requirements on a particle class besides that it should be a sub-class of GenericParticle. It must of course define creation and destruction methods. It must also implement a method for reproduction, which should be declared virtual and be called Copy(). This is because the ParticleFactory which will handle the production of particles does not have any a priori knowledge of exactly what the particles look like, but is only provided at run-time with one template particle of each class. Hence the Copy() method must be provided in order that the ParticleFactory may handle the reproduction.

In fig. 3 we show the declaration of the GenericParticle class. As seen the only actual instance variables besides the inherited four-momentum is the mass and helicity[#2] all other information is stored centrally by the ParticleFactory but can be easily accessed by the pointers to the decayTable and genericData, etc. The methods AddParameter, SetParameter, AddSwitch, etc. are explained in section 2.4.

The simplest example of a particle class is the Lepton class as shown in fig. 4. As no new instance variables are introduced the creation methods just passes its arguments along to the GenericParticle together with the generic name "lepton". Also the Copy() method is quite

[#2] Spin and helicity are actually not used in any way in this version of MC++ but are included for future use.

```
class GenericParticle : public fourMom {
 public:
   GenericParticle();
   GenericParticle(String &, GenericData *, DecayList *);
   ~GenericParticle();
   String &GenericName();
   String &Name();
   int Type();
   int ICharge();
   float Charge();
   int ISpin();
   float Spin();
   float Mass();
   float MassWidth();
   float MassCut();
   virtual float ThisMass();
   boolean HasDecayed();
   ParticleFactory &Factory();
   virtual TransMat &Boost(double, double, double);
   virtual TransMat &RotatePhi(float);
   virtual TransMat &RotateTheta(float);
   virtual void Transform(TransMat&);
   void AddChild(GenericParticle *);
   virtual void SetMass();
   void AddDecayChannel(DecayEntry *);
   void AddParameter(String &, String &, float *, float,
                     float *, float *);
   void AddParameter(String &, String &, int *, int,
                     int *, int *);
   void SetParameter(String &, float);
   void SetParameter(String &, int);
   void AddSwitch(String &, String &, int *);
   void AddSwitchOption(String &, String &, int);
   void SetSwitch(String &, int);
   void PrintSwitch();
   ParticleList &Select(Selector);
   ParticleList &Decay();
   virtual void Print();
   virtual GenericParticle *Copy();
 protected:
   ParticleFactory *factory;
   GenericParticle *parent;
   ParticleList *childList;
   DecayList *decayTable;
   GenericData *genericData;
   String genericName;
   boolean hasDecayed;
   float mass;
   int helicity;
};
```

Fig. 3. The class declaration of GenericParticle. Its most important member variables include a pointer to a GenericData structure, which holds the information common to all instances of a particular particle type, e.g. a pi0, a pointer to a list of decay products and the mass and helicity.

```
class Lepton : public GenericParticle {
 public:
   Lepton(GenericData *, DecayList *);
   GenericParticle *Copy();
};


Lepton::Lepton(GenericData *gd, DecayList *tab)
  : GenericParticle("Lepton", gd, tab) {}


GenericParticle *Lepton::Copy(){

   Lepton *newp = new Lepton(genericData,
decayTable);

   newp→mass = mass;
   newp→helicity = helicity;
   newp→hasDecayed = hasDecayed;
   newp→parent = parent;
   newp→factory = factory;

   return newp;
}
```

Fig. 4. The Lepton class, a simple example of how to derive new particle types from the generic particle class. The figure also shows the definition of the constructor and the Copy() function.

simple, just creating a new Lepton and copying over the variables.

A more complicated particle class is the QCDString class shown in fig. 5. A QCDString is a Lund-type string and may contain a number of partons contained in the list pl. In MC++ partons can in principle only exist within a QCDString. That is only colour singlet objects are treated as true particles. This may be inconvenient for models where partons are e.g. allowed to fragment independently, but having partons as independent particles would make MC++ impossible to use for e.g. string fragmentation models.

As QCDString is a rather complicated particle it redefines the methods for printing. It also introduces new methods for adding partons to the parton list and to retrieve the parton list from the QCDString object. The Copy() method is also different from that of the Lepton in that it copies also all the partons.

```
class QCDString : public GenericParticle {
 public:
   QCDString();
   QCDString(GenericData *, DecayList *);
   virtual GenericParticle *Copy();
   PartonList *PList();
   void AddParton(Parton *);
   virtual void Print();
   virtual void PrintAll(String &prefix = "1");
   virtual void SetMass();
   virtual void SetMass(float);
 private:
   PartonList pl;
};
```

Fig. 5. The `QCDString` class, a more elaborate example of a derived particle class. The `QCDString` defines, in addition to the base class members, a list of partons and a method to add new partons to the list.

```
class Decayer {
 public:
   Decayer(String &, int);
   virtual int Decay(GenericParticle *, ParticleList *);
   virtual boolean IsAllowed(GenericParticle*,
ParticleList*);
   String &GenericName();
   String &Name();
   int &Id();
   virtual Decayer *Copy();
   void SetFactory(ParticleFactory *);
   void AddParameter(String &, String &, float *,
                     float, float *, float *);
   void AddParameter(String &, String &, int *,
                     int, int *, int *);
   void SetParameter(String &, float);
   void SetParameter(String &, int);
   void AddSwitch(String &, String &, int *);
   void AddSwitchOption(String &, String &, int);
   void SetSwitch(String &, int);
   void PrintOptions();
 protected:
   ParticleFactory *factory;
   String genericName, name;
   int id;
   ParameterList parameters;
   SwitchList switches;
};
```

Fig. 6. Class declaration of `Decayer`. This class defines the interface to a decayer. It also defines a uniform way to access parameters and switches through the functions `AddParameter`, `SetParameter`, etc.; these functions have the same form as for the `GenericParticle` class.

2.2. The `Decayer` class

As noted above, the actual method or algorithm to decay a generalized particle is not implemented in the particle classes themselves, but rather in the class `Decayer`. The definition of the base class `Decayer` is shown in fig. 6. The purpose of this base class is to define the interface to a general decayer, and the derived decayers are then in this context a specification of the generic decayer, implementing a specific decay model or method. The base class also declares and defines methods to add and change parameters and switches of the decayer.

In this way the `Decayer` base class defines a simple and uniform way for the user to access parameters and switches, the same for all decayers, this should ease the construction of a interactive front end to the MC++ kernel.

As seen in fig. 6, the member variables of the base class includes a pointer to the factory where the decayer is stored, two strings for the name and the generic name of the decayer, respectively. The generic name denotes a specific class of the decayer, whereas the name denotes a specific instance of a decayer.

The base class also has a list of parameters and a list of switches. Among the member functions there are three most important functions `IsAllowed()`, `Decay()` and `Copy()`, these functions are declared virtual and the base class only defines them to return an error message as these functions should be defined by the derived classes. The `IsAllowed()` functions returns true or false if the particle pointed to by the first argument is allowed to decay into the list of particles pointed to by the second argument. The function `Copy()` is expected to return an explicit copy of the decayer.

The other functions are both declared and defined by the base class, they include methods to add parameters and switches and to set these. The parameters and switches are member variables of a decayer, and they are added to the parameter list and switch list in the constructor of the decayer to make sure that they can be accessed through the member functions of the base class.

```
class TwoBodyDecayer : public Decayer {
 public:
   TwoBodyDecayer(String &, int);
   virtual boolean IsAllowed(GenericParticle *,
ParticleList *);
   virtual Decayer *Copy();
   virtual int Decay(GenericParticle *, ParticleList *);
};

boolean TwoBodyDecayer::IsAllowed(GenericParticle
*parent,
                                        ParticleList
*children) {

   if ( children→Length() == 2 )
      return true;
   else
      return false;
}

Decayer *TwoBodyDecayer::Copy(){

   TwoBodyDecayer *dec = new TwoBodyDecayer("",
0);

   dec→factory = factory;
   dec→name = name;
   dec→id = id;

   return dec;
}
```

Fig. 7. The `TwoBodyDecayer`. The figure shows the declaration of the class, in this case there are no additional member variables, but rather it only defines the functions `IsAllowed()`, `Copy()` and `Decay()`. The definitions of `IsAllowed()` and `Copy()` are shown.

2.2.1. A twobody decayer

As a first simple example of a decayer we show how a twobody decayer is implemented. The declaration of the class is shown in fig. 7. This class does not declare any new member variables, but instead only defines the three mandatory functions `Copy()`, `Decay()` and `IsAllowed()`. The `IsAllowed()` function, is simply defined to return "true" if the number of children are two, and "false" otherwise. The `Copy()` function creates a new instance of the class `TwoBodyDecayer` to ensure that the proper definition of the member functions gets called when the this copy is used to decay particles.

```
int TwoBodyDecayer::Decay(GenericParticle *parent,
ParticleList *children) {

   GenericParticle* ch1 =
factory→GetA(children→First());
   GenericParticle* ch2 =
factory→GetA(children→Second());

   if(ch1→Mass() + ch2→Mass() ≥ parent→Mass()) {
      delete ch1;
      delete ch2;
      return 0;
   }

   Random &random = factory→rnd;

   float m12 = ch1→p2();
   float m22 = ch2→p2();

   float e = parent→Mass();

   float e1 = (e*e - m22 + m12)/(2*e);
   float pz = sqrt(e1*e1 - m12);

   ch1→SetFourMom(0.0, 0.0, pz, e1);
   ch2→SetFourMom(0.0, 0.0, -pz, e-e1);

   TransMat m;
   m.rotateTheta(acos(random.Flat(-1.0, 1.0)));
   m.rotatePhi(random.Flat(2*PI));
   m.boostFromCMOf(parent);

   ch1→Transform(m);
   ch2→Transform(m);

   ch1→SetParent(parent);
   ch2→SetParent(parent);

   parent→AddChild(ch1);
   parent→AddChild(ch2);

   return 1;
}
```

Fig. 8. Definition of the `Decay()` method of the `TwoBodyDecayer`.

Finally, the method `Decay()` is shown in fig. 8. This function performs the actual decay of the particle. The code is hopefully simple enough so that even someone with little knowledge about C++ can understand it.

```
class AriadneDecayer : public Decayer {
 public:
   AriadneDecayer(String &, int);
   virtual boolean IsAllowed(GenericParticle *,
ParticleList *);
   virtual Decayer *Copy();
   virtual int Decay(GenericParticle *, ParticleList *);
 private:
   float lambda_QCD;
   float alpha_s;
   float pt_cutoff;
   float pow_sup;
   float soft_sup;
   int maxEmissions;

   int parmStrategy;
   int fragment;
   int phaseSpace;
   int alphaStrong;
   int recoils;
   int extPartons;
   int invPt;
   int alphaStrongArg;
   int dumIMin, dumIMax, dumIZero;
   float dumFMin, dumFMax, dumFZero;
};
```

Fig. 9. Class definition of AriadneDecayer. This class defines in addition to the functions IsAllowed(), Copy() and Decay(), a set of member variables to be added to the decayers parameter and switch lists by the constructor of the class.

2.2.2. *The Ariadne decayer*

The next example shows how to encapsulate an existing Fortran Monte Carlo into an MC++ decayer, in this example Ariadne 3.1 [4]. The class declaration is shown in fig. 9. Apart from the functions Copy(), IsAllowed() and Decay(), it declares several variables that are added to the parameter and switch lists.

In fig. 10 part of the constructor of AriadneDecayer is shown. Here the member variables are added to the list of parameters or the list of switches; these variables can then be changed through member functions, defined in the base class Decayer, or by reading new data from a file using the ReadFrom function of ParticleFactory.

The implementation of the Decay() function is seen in fig. 11. It first adds the particle data into the lujets common block, calls the ariadne_() function to decay the particle, in this case to perform the cascade of a $q\bar{q}$-dipole according to the dipole model. It then extract the cascaded strings from the event record, creates the corresponding MC++ particles and adds them to the particles child list.

```
AriadneDecayer::AriadneDecayer(String &_name, int _id)
  : Decayer("Ariadne", _id) {

   name = _name;

   AddParameter("lambda_QCD",
                 "The momentum scale in alpha strong",
                 &lambda_QCD, 0.25, &dumFZero,
&dumFMax);


   AddSwitch("recoils",
             "Quark recoil strategy", &recoils);
   AddSwitchOption("recoils",
                   "All partons treated the same", 0);
   AddSwitchOption("recoils",
                   "Quarks takes all the recoil if
                    they are not extended", 1);

   SetSwitch("recoils", 1);

.
.
.

}
```

Fig. 10. Definition of constructor of AriadneDecayer. As mentioned in the text this function adds to the list of parameters and switches, those member variables that should be accessible to the user.

2.3. *The ParticleFactory class*

In order to administrate the production of particles in a simple manner, we have designed the class ParticleFactory, see fig. 12. This class contains information about all particles and decayers. It has, through its member functions, facilities to read data for parameters and switches from files, and thus enable the user to change the default behaviour of particles and decayers. The ParticleFactory class also provides methods to add new particles and decayers to the factory, so that the model developer can easily extend the MC++ kernel with new models, without being forced to make changes to the ParticleFactory class itself.

```
int AriadneDecayer::Decay(GenericParticle *parent,
                          ParticleList *children){

  QCDString *st = (QCDString *)parent;

  VAR[0] = lambda_QCD;

  :

  KAR[2] = recoils;

  Parton *p;
  int i = 0;
  while( (p = (st→PList())→Nth(i+1)) ≠ 0 ){
     K[0][i] = 2;
     K[1][i] = p→Type();
     K[2][i] = K[3][i] = K[4][i] = 0;
     P[0][i] = P[1][i] = 0.0;
     P[2][i] = p→pz();
     P[3][i] = p→e();
     P[4][i] = p→Mass();
     V[0][i] = V[1][i] = V[2][i] = V[3][i] =
V[4][i] = 0.0;
     i++;
  }
  K[0][i-1] = 1;
  N = i;

  ariadne_();

  i = 0;
  while ( i < N ){
     QCDString *child =
factory→GetA("cascaded_string");
     do {
        p = factory→GetA(K[1][i]);
        p→SetFourMom(P[0][i], P[1][i], P[2][i],
P[3][i]);
        child→AddParton(p);
     } while ( K[0][i++] == 2);

     parent→AddChild( child );
  }
  return 1;
}
```

Fig. 11. Definition of the `Decay()` member function of `AriadneDecayer`. This function adds the data of the parent to the `lujets` event record, calls the Fortran function `ariadne_` to perform the cascade, and then extract the resulting strings from the event record and creates the necessary MC++ particles.

```
class ParticleFactory {
 public:
   Random rnd;
   ParticleFactory(String fileName = "pdg.ptab");
   void *GetA(String &name);
   void *GetA(int type);
   void *GetDecayer(String &name);
   void *CopyDecayer(String &name);
   void *GetDecayer(int type);
   void ReadFrom(String &);
   void AddParticle(GenericParticle *p);
   void AddParticleTemplate(GenericParticle *p);
   void AddDecayer(Decayer *dec);
 private:
   ParticleList particle_templates;
   ParticleHashTable pTable;
   DecayerList decayers;
};
```

Fig. 12. Class declaration of `ParticleFactory`. This class is used to maintain data on the articles and decayers, and to provide the user with a simple way to access these particles and decayers.

The constructor of the class takes as argument the filename of a setup file containing data on the particles and decayers; the format of the setup files is described below. This file is intended to contain default data on particles and decayers independent of the particular physics model the user wants to work with, so the user should not make any changes to this file. Instead, to be able to change the default data, the `ParticleFactory` class has a member function `ReadFrom(String &filename)` that reads the file with the name given as argument and makes the necessary changes to the factory.

To get a generalized particle from the factory, the user simply calls the member function `GetA()`, where the argument can be either a string with the name of the particle, or an integer corresponding to the type of the particle. There is a third version of the function `GetA`, which takes a pointer to a `GenericParticle` as argument, this version simply returns a copy of its argument and is typically used within a decayer object as seen in fig. 8.

Similarly, the user can retrieve a decayer from the factory in order to change parameters or switches of the decayer. This is done with the functions `GetDecayer()` and `CopyDecayer()`, where the argument obviously is the name of the decayer.

Apart from extracting particles, the `ParticleFactory` has member functions to allow the user to easily add new particles and particle classes. This is done with the functions `AddParticle` and `AddParticleTemplate`, respectively. A particle template is just an instance of a particle class. The `ParticleFactory` class maintains a list of these templates, one for each particle class, which are needed in order to make sure that a particle is created with the correct constructor. For example, when the user wants to add a particle of the `Lepton` class to the factory, for instance with a command in a setup file, see below, the factory first gets a pointer to the lepton template to get an explicit copy of the lepton particle, adds the data about the particle to be added in the factory, and finally adds the new particle to a hash list of particles. This is to ensure that, e.g. a particle of the class `Lepton` has been created with `Lepton *p = new Lepton()` so that the functions defined in the `Lepton` class are called when the particle is used.

The specific particles, e.g. `pi+`, `K-` etc., are, as mentioned, stored in the factory in a hashed list in order to make the retrieval of particles efficient and fast. The hash table in the factory provides methods to lookup the particles either using the name of the particle or using the particle id.

2.4. Utilities

In this section we describe briefly some of the other classes used in the toolkit. These include a random number generator and classes for parameters and switches. The toolkit uses more classes than we describe in this paper, but since the intent is to explain the idea behind the project, we refer the interested reader to the code, which is available as described below.

2.4.1. Random

The MC++ toolkit is by default supplied with a random number generator, see fig. 13. The generator uses the algorithm described in ref. [12],

```
class Random {
    float u[97];
    float c,cd,cm;
    long i97,j97;
    float ran;
    int set;
    int dump_state(FILE *);
  public:
    Random( long seed = 19780503);

    int AppendState( char *name = "random.state");
    int WriteState( char *name = "random.state");
    int ReadState( int rec = 1, char *name =
"random.state");

    inline float Flat();
    inline float Flat( float );
    inline float Flat( float, float );
    inline float Exp();
    inline float Exp( float );
    inline float Gauss();
    inline float Gauss( float, float );
    inline float BreitWigner( float, float );
    inline float BreitWigner( float, float, float );
    inline float BreitWignerM2( float, float );
    inline float BreitWignerM2( float, float, float );
};
```

Fig. 13. Class declaration of `Random`. This class provides the user with a set of commonly used distributions of random numbers.

and it is the same as the one supplied in JETSET 7.3. This generator has a very long period $\mathcal{O}(10^{43})$ an has the ability to generate $\mathcal{O}(10^9)$ disjoint sequences of random numbers.

To provide the user with random numbers with different distributions, the `Random` class has a set of member functions to supply the user with this functionality, as seen in fig. 13. The distributions provided are flat, exponential, Gaussian and Breit–Wigner distributions. The different calling sequences for the member functions give the user distributions with different mean and variance. The functions with no arguments give distributions with unit mean and variance.

The Breit–Wigner distributions are given by the following equations:

$$P(x)\,\mathrm{d}x = \frac{\mathrm{d}x}{(x-x_0)^2 + \Gamma^2/4} \qquad (1)$$

and

$$P(x)\,\mathrm{d}x^2 = \frac{\mathrm{d}x^2}{(x^2-x_0^2)^2+x_0^2\Gamma^2}, \qquad (2)$$

respectively.

2.4.2. Parameters and switches

Most decayers and particles will have parameters and switches to describe or determine the behaviour of the decayer or the particle. To have an unified and consistent way of handling this functionality, we have implemented the classes `Parameter` and `Switch`. The `Parameter` has member functions to allow for both integer and floating-point values.

The class declares three pointers, one pointing to a member variable in a decayer or particle that should be accessible to the user, the other two pointing to the minimum and maximum value of the decayer. The reason to have pointers to the minimum and maximum value is to allow the system to check that a user does not assign a value to a parameter outside the bounds of the parameter in question, furthermore the bounds of a parameter can depend on the values of other parameters so that the min or max pointers can be made to point to other parameters.

In addition the class declares a variable to hold the default value of the parameter, and two strings, one for the name of the parameter and one for a short description.

The `Switch` class, similarly declares a pointer to the switch, an integer member variable in a decayer or a particle, to give the user the possibility to change the value of a switch in a simple and consistent manner. The class also declares a list of options, where each option contains a value and a string with a description of the corresponding option.

3. How to use MC++

In fig. 14 we show a simple main program for generating e^+e^- events at LEP energies with MC++. First of all the `ParticleFactory` is created. At this point the factory creates template objects for each particle and decayer class which

```
#include "pfactory.h"
#include "genparticle.h"
#include "ariadne.h"
#include "selectors.h"

main(int argc, char* argv[]){

  ParticleFactory factory("pdg.ptab");

  InitAriadne(factory,"ariadne.ptab");

  GenericParticle *p;

  if( p = factory.GetA("e+e-collision") ){
    p→DecayAll();
    p→PrintAll();
  }

  ParticleList &list = p→Select(IsStable);

  list.Print();

}
```

Fig. 14. A simple example of a main program using the MC++ toolkit, showing how to generate a e^+e^- event.

are a part of MC++, then it reads a default setup file which describes all the actual particles that should be created and how they decay, typically using information available from the PDG table of particle properties.

After the factory is created the models that should be used are loaded. This is done with a function that must be defined for each model, which takes a `ParticleFactory` as an argument (`InitAriadne`). In this function, template decayer and particle objects of all classes that are special for this model are loaded into the `ParticleFactory`. Also the `ParticleFactory` is told to read from a setup file to set up all decay modes etc. that are particular to this model.

The `ParticleFactory` is now ready to produce events according to the model chosen. It is of course still possible to read more setup files containing the users preferred changes to the model chosen.

3.1. Setup files

The setup files are clearly important parts of the MC++ structure. They are simple ASCII files containing commands which can be understood by the `ParticleFactory`. The command syntax looks like this.

- `add p gName pName pNr mass massWidth massCut charge spin lifeTime`
  adds a particle of class `gName` to be called `pName` and numbered `pNr`, giving it a mass and a width cut off at `massCut`, a charge a spin and a `lifeTime`.
- `add c pNr brat dName N pNr1 ... pNrN`
  adds a decay channel to particle `pNr` with a branching ratio `brat` for decaying into `N` particles with numbers `pNr1` to `pNrN`.
- `sets [p|d] [pName|dName] sName value`
  sets the switch called `sName` to `value` for a particle or a decayer called `pName` or `dName`.
- `setf [p|d] [pName|dName] fName value`
  sets the floating-point valued parameter called `fName` to `value` for a particle or a decayer called `pName` or `dName`.
- `seti [p|d] [pName|dName] iName value`
  sets the integer valued parameter called `iName` to `value` for a particle or a decayer called `pName` or `dName`.
- `add d gName dName dNr`
  adds a new instance of a decayer of class `gName` naming it `dName` and giving it a number `dNr`.

3.2. Communication with other programs

MC++ so far does not have any facilities for event analysis, indeed all you can do at the moment is to perform the decay and, as is shown in fig. 14, to print out the decay chain and do things like extracting a list of stable particles and print them. This is of course not very useful.

Ideally, the `ParticleFactory` of MC++ would simply be incorporated into a detector simulation program, preferably also written in C++. In this way the particles produced by MC++ could be propagated through the detector until they decay, whereupon its children can continue to propagate. In this way the detector simulation program does not have to know anything about how particles are produced and decayed as all this information is contained in the particle objects, but you still get the very natural picture of events that are actually produced "in the detector". A project for developing a detector simulation program, called Gismo [13] has started and some work has been done to make MC++ compatible with this program.

Work has also started to enable MC++ to output so called Cheetah records [14]. Cheetah is a platform-independent data management system which would allow MC++ to communicate with other programs by writing out the event information to a file (or a pipe) which is read by e.g. an analysis program.

3.3. Developing new particles and decayers

When developing new models with MC++ there are a few restrictions which have already been mentioned above.

All new decayer classes must be derived (directly or indirectly) from the generic decayer class `Decayer`. The new class must overload the member functions `Decay()`, `IsAllowed()` and `Copy()`. All new instance variables that are to be available for manipulation by the user should be inserted in the `Parameter` or `Switch` lists.

In the same way all new particle classes must be derived from the `GenericParticle` class and the member function `Copy()` must be overloaded. Optionally, also the methods for printing, boosting, rotating and setting of mass can be overloaded. Again all new instance variables that are to be available for manipulation by the user should be inserted in the `Parameter` or `Switch` lists.

It is important to note that the models do not have to be completely written in C++. As a matter of fact all the models that have been adapted to MC++ so far are "old" FORTRAN programs, which have simply been "encapsulated" into `Decayer` classes. So as far as MC++ is concerned, FORTRAN is not "dead", it has simply been put somewhat in the background to do what it is best at: to do the heavy numerical work.

4. The future of MC++

As mentioned in the introduction, this version of MC++ is not complete in any way. Especially unsatisfactory is the `Collision` class. The aim is to set up a number of classes defining the "decay chain" from a collision through the hard interaction, initial- and final-state parton showers and hadronization for all kinds of collisions in such a way that all present models can use them. Also classes for structure function parametrizations should be constructed. This, of course, requires a lot of work and it is important to have a discussion with the different MCEG authors to be able to agree on a common structure.

So far MC++ is not equipped with any user interface. However, the whole structure is set up to be easily accessed by a graphical user interface and work has started to build an application based on the NeXTStep window system. This would enable the user to more easily manipulate setup files which is clearly essential for the usefulness of MC++. Also an interface for X-Windows is planned.

To summarize, we have found that it is, if not simple so at least possible, to formulate the event generating chain in high-energy collisions using OOP techniques. Although this version of MC++ is still at an experimental stage we have found that the structure is general enough to make it possible to use many different models within its framework (besides the JETSET and Ariadne programs mentioned in this paper, also parts of the HERWIG program has been successfully interfaced to MC++ [16]).

We will continue developing MC++ and we would like to encourage anyone who is interested to contact us so that the base for this project can be broadened.

Appendix. Technical Information

The zeroth version of the MC++ toolkit is available through anonymous ftp from `thep.lu.se` (`130.235.92.57`) as the files
`pub/MCPP/MC++0.0.tar`, or
`pub/MCPP/MC++0.0.tar.Z`,
the former is a `tar` archive file and the latter is a compressed version of the `tar` file. If you do not have access to ftp or to a `tar` archive program, please contact one of the authors to work out alternative possibilities for distribution.

Required for compiling the toolkit is a C++ compiler implementing version 2.0 or later of the C++ language. Except for a iostreams library it does not require any additional libraries. A small library of character string and list classes, kindly provided by Dag Brück at the Department of Automatic Control, Lund Institute of Technology, has, however, been included in the MC++ toolkit.

So far the toolkit has been compiled on the following platforms:

- NeXTstation with the native C++ compiler plus libg++ version 1.39.
- DECstation 3100 with g++ version 1.39 and libg++ version 1.39.
- Apollo 425t workstation with Domain/C++ version 2.1.0.
- VAX/VMS with g++ version 1.39.

The platform dependencies of MC++ is handled by the use of macros in the source files. The macros are defined in the top level Makefile in order to make the compilation simple, so the user should only need to edit the top level Makefile. This Makefile defines a macro called DEFS which should be assigned the definitions needed for the specific machine type and compiler. The comments in the top level Makefile should give the necessary informations. The distribution includes sample Makefiles for the different platforms. For more detailed information on the installation procedure we refer to the `README` file in the distribution.

We have tried to write the code in a consistent and uniform way to be able to use a utility written in AWK called `classdoc` [15], which translates C++ definition files into a more readable form, using a UNIX manual page like format.

The distribution also contains a couple of small test programs for generating events using the Ariadne and JETSET models. To run these you will need the FORTRAN source for these programs, available from the CERN program library.

References

[1] G. Marchesini and B.R. Webber, Nucl. Phys. B 310 (1988) 461.
I.G. Knowles, Nucl. Phys. B 310(1988) 571.

[2] F.E. Paige and B.S. Protopopescu, in: Proc. Snowmass Summer Study 1986 (QCD184:S7:1986) p. 320.

[3] B. Bambah et al., QCD Generators for LEP, CERN-TH.5466/89.
T. Sjöstrand, Comput. Phys. Commun.39 (1986) 347.
T. Sjöstrand and M. Bengtsson, Comput. Phys. Commun. 43 (1987) 367.

[4] L. Lönnblad, ARIADNE 3 – A Monte Carlo for QCD cascades in the colour dipole formulation, Lund Preprint LU TP 89-10 (1989).

[5] A. Kwiatkowski, H. Spiesberger and H.J. Mohring, HERACLES: an event generator for ep physics at HERA energies including radiative processes, version 1.0, DESY preprint DESY-90/041 (April 1990).

[6] N. Magnussen, Results from the working group for event generators, in: Proc. Workshop Physics at HERA, Hamburg, October 1991, to be published.

[7] B. Andersson and G. Gustafson,Phys. C 3 (1980) 223.
B. Andersson, G. Gustafson, G. Ingelman and T. Sjöstrand, Phys. Rep. 97 (1983) 31.

[8] B. Bambah et al., QCD generators for LEP, CERN-TH.5466/89.

[9] R. Blankenbecler and L. Lönnblad, Particle production and decays in an object oriented formulation, Lund Preprint LU-TP 91-19 and SLAC preprint SLAC-PUB 5648; Particle World, to be published .

[10] P. Kunz, Object oriented programming, SLAC-PUB-5629, (August 1991); in: Proc. 1991 CERN School of Computing, Ystad, Sweden, September 1991, to be published.

[11] B. Stroustrup, The C++ Programming Language, 2nd ed. (Addison–Wesley, Reading, MA,1991).
S. Lippman, C++ Primer, 2nd ed. (Addison–Wesley, Reading, MA, 1991).

[12] F. James, Comput. Phys. Commun. 60 (1990) 329.

[13] W.B. Atwood, T.H. Burnett, R. Cailliau, D.R. Meyers and K.M. Storr, Gismo: application of OOP to HEP detector design, simulation and reconstruction, in: Proc. Computing in High Energy Physics 1991, Tsukuba, March 1991.

[14] P. Kunz and G.B. Word, The Cheetah data management system, SLAC-PUB-5450 (March 1991).

[15] D. Brück, Classdoc utility, Department of Automatic Control, Lund Institute of Technology, P.O. Box 118, S-221 00 Lund, Sweden.

[16] M. Seymour, in preparation.