УДК 539.12.162.8

AN OBJECT-ORIENTED FRAMEWORK FOR THE HADRONIC MONTE-CARLO EVENT GENERATORS

***N.Amelin*[1], *M.Komogorov*[2]**

We advocate the development of an object-oriented framework for the hadronic Monte-Carlo (MC) event generators. The hadronic MC user and developer requirements are discussed as well as the hadronic model commonalities. It is argued that the development of a framework is in favour of taking into account of model commonalities since common means stable and can be developed only at once. Such framework can provide different possibilities to have user session more convenient and productive, e.g., an easy access and edition of any model parameter, substitution of model components by the alternative model components without changing the code, customized output, which offers either full information about history of generated event or specific information about reaction final state, etc. Such framework can indeed increase the productivity of a hadronic model developer, particularly, due to the formalization of hadronic model component structure and model component collaborations. The framework based on the component approach opens a way to organize a library of the hadronic model components, which can be considered as the pool of hadronic model building blocks. Basic features, code structure and working examples of the first framework version for the hadronic MC models, which has been built as the starting point, are shortly explained.

The investigation has been performed at the Laboratory of High Energies, JINR.

Объектно-ориентированный фреймворк для создания и использования генераторов событий взаимодействия частиц и ядер

Н.Амелин, М.Комогоров

Обсуждается разработка пакета программ (объектно-ориентированный фреймворк) для создания и использования генераторов событий взаимодействий частиц и ядер. В частности, рассматриваются требования, возникающие при разработке и использовании моделей такого типа, общие свойства присущие моделям данного типа, а также их реализация посредством современной компонентной технологии программирования. Такой подход дает пользователю, работающему посредством единообразного интерфейса, ряд удобств, например: редактирование любых параметров

[1]Nikolai.Ameline@cern.ch
[2]max@sunhe.jinr.ru

модели, замещение одних компонент другими, конфигурирование выходных данных, которые содержат либо полную информацию об истории генерации события, либо только специфическую информацию о конечном состоянии взаимодействия. Компонентный подход существенно увеличивает производительность разработчика моделей, так как все составляющие пакета реализованы в виде набора формализованных фреймов, описывающих различные типы компонент. Он позволяет организовать хорошо структурированную библиотеку таких компонент с удобной системой навигации по ним, что обеспечивает универсальный и гибкий механизм построения сложных многокомпонентных моделей. В данной публикации также представлены основные составляющие первой версии пакета, приведены примеры его использования. Эта версия может быть рассмотрена как отправная точка для построения более сложной и мощной системы.

Работа выполнена в Лаборатории высоких энергий ОИЯИ.

1. INTRODUCTION

The object-oriented approach based on the C++ can be adopted to write the hadronic MC event generators codes. Such approach has many advantages as compared with traditional procedural coding (see [1]). Even more we advocate to build an object-oriented framework [2], since we have observed many commonalities for the hadronic MC models as well as for their usage and for their development. Framework approach is more justified, when the list of models chosen for development is very large and potentially can be increased.

A framework [2] can be considered as well-documented thematic collection of software to build related applications. It outlines the main architecture for the application to be developed. The successful framework should not only support needed features and provide default implementation and built-in functionality as much as possible, but it should also allow an easy modification and extension of the built-in functionality. The main goal of any framework is a reusibility. The software developer should be able to reuse written code (e.g., classes from the framework libraries) and the design of a framework. A framework design is closely connected with design patterns used to document certain elements of it. A design pattern is a concise definition of a technique that demonstrates some successful solution for particular coding problem. Particularly, the factories and the proxies design patterns (see book [3]) have been applied in our framework version.

We assume that our object-oriented framework for the hadronic MC generators will be useful for two categories: the model users and the model developers (the advanced users). We consider a hadronic model user as a person who interacts with the framework by means of a user interface without writing and modifying of the model codes. A hadronic model developer is assumed to work with the framework on the level of internal framework interfaces. A developer needs the knowledge of the framework structure and libraries as well as the knowledge of C++ language.

The main idea of our framework is to adopt the component approach. We have in mind that such framework can be the base to build an extending library of the model components and the model algorithms. It can allow us to extract model components from the model components library pool and to compose them into the powerful physical models.

Below we would like to outline the hadronic model commonalities as well as the requirements for hadronic model package from the hadronic model user and developer. Then we

want to explain our framework basic ideas and shortly describe the possible user and developer sessions. To understand the framework performance there are several important parts of its architecture that need to be explained as well as interaction of these parts. Thus we would like to provide short explanation of the model components and their structures, the framework control subsystem (the dispatcher) and its work, the set of classes for the parameter and input maps, the data transfer class library, the output subsystem with the data exchange format and the data file structure, etc. Finally, we offer several examples to illustrate the framework work.

2. COMMONALITIES OF THE MC HADRONIC MODELS

The MC hadronic models can be used as the hadronic event generators with the main goal to study hadronic collision phenomena as well as the source of information about hadronic collision final states with the aim to utilize this information.

Even taking a fast look at the MC hadronic models one can see that they have much in common. First of all they are phenomenological models having large amount of model parameters. We can specify these parameters as the physical parameters (hadronic model tuning constants) and hadronic model configurators. The first type of parameters gives a possibility of changing hadronic model results. They operate similarly as the physical input. This type of parameters fulfils very important job to store physical information about hadronic processes. The second type of parameters also offers a possibility of changing hadronic model results, but by changing the application logic of a physical algorithm.

Additionally to the parameters much more information should be provided for any hadronic model. For example, the information about physical properties of particles: quark contents, electric charges, masses, decay branchings, etc., is required. The information about the physical properties of stable and excited nuclei: binding energies, spins, level density parameters, fission barrier heights, etc., is also required. Usually such information is needed in the read-only mode.

The MC hadronic models act in a similar way. They either convert an input into an output or they use an input to give an answer to the user request. The input can be the characteristics of particles: hadrons, partons, gammas, etc., or characteristics of nuclei: stable nuclei, excited fragments, etc. The output can be again a set of particles: hadrons, partons, gammas, etc., or a set of nuclei: stable nuclei, excited fragments, etc. Acting so any MC hadronic model has to deal with four vectors (energy-momentum and time-position) and their transformations. Any hadronic model somehow handles the n-body kinematics.

The most of hadronic models are the multicomponent models. A multicomponent model includes other models as additional or alternative model components and has complicated execution flow. Especially for application purposes a user needs the set of hadronic models to obtain proper description of the hadronic reaction final states [4].

Practically all hadronic models are complicated numerical models. For them it is not always trivial to separate «physics» from «algorithm», i.e., to separate the physical input, physical parameters, etc., influence on the simulation results from the chosen numerical algorithm influences. It is also often, when the same numerical algorithm can be reused within models describing different physical phenomena, e.g., the decay of resonances and the excited nuclei according to the relativistic phase space, the elastic scattering of partons,

hadrons and nuclei, the search collision and decay algorithm for the hadron transport model and the parton transport model, etc.

The different kinds of errors can be occurred during hadronic model initialization or model runtime. The source of errors can be due to the inconsistent user input as well as due to the complexity of numerical algorithms. The last situation is an often unexpected situation.

Any hadronic model is required to produce different physical outputs, which should be analyzed. The output can be only specific information about hadronic reaction final states or complete information about the history of a generated event.

The above list of commonalities can be more extended. For example, besides kinematics all hadronic MC models deal with random sampling of variables according to the different probability distributions, i.e., large set of the random number generators are required. To perform numerical operations, different mathematical utilities: equation solvers, integrators, interpolators, etc., are needed to be employed. However, it is already clear that a hadronic model developer should take into account these commonalities by either common code structure or common used methods or common implementations, etc.

3. TYPICAL REQUIREMENTS FROM HADRONIC MODEL USER AND DEVELOPER

Any user of the hadronic models needs model descriptions, i.e., the reaction mechanisms, model inputs, model parameters, used numerical algorithms, etc., should be described. Such descriptions should be accompanied by needed examples to demonstrate model usage.

The different usage strategies provide different user requirements for hadronic model package. A user performing theoretical or experimental study of hadronic collision needs a possibility to «play» with chosen model, e.g., the possibility to visualize and change model parameters, to configure model or model component, to choose an alternative model component, to customize hadronic model output. Thus, the mechanism to check consistency of the user alterations should be provided. For this type of user, run control requires to have hadronic model runtime information and exception handling mechanism.

Another type of user or applied user are mostly interested in the generated event itself. The configured for a given type of hadronic reaction model with default values of parameters should be offered for the applied user. The output information should be reduced until required minimum and presented in the required form.

Of course, both types of users need to have much more, e.g., simple and self-explanatory mechanism for hadronic reaction input preventing from errors due to the inconsistent input, the mechanism to throw out event due to the possible severe errors at runtime, the analysis and visualisation tools are also required to analyze the generated output, etc. Thus, for any kind of users a user session should be convenient and productive.

The model developers may want to rebuild an existing hadronic model with aims to extend the range of its applicability, to increase its predictive power, etc. They may want to incorporate an existing («foreign») hadronic model for a cooperative work with other existing models. Also they may want to build a new hadronic model running standalone or in collaboration with other models. The enumerated situations are the primer tasks of a hadronic model developer. But in reality to satisfy the user requirements he or she should realize the user interfaces and even do much more. The interfaces between created hadronic model code

and the outside world, e.g., an operational system, are also needed to be realized as well as different adapters, if he or she wants to use an external packages, e.g., to visualize and analyze the data. A developer should have in mind its hadronic model package portability. The package portability means not only the portability in the sense to work on different computer platforms, but also, it is the possibility for hadronic package to work within another package. A hadronic model developer should facilitate the future developers tasks as well.

In the conventional approach for the developing of the hadronic MC models a developer or several developers are working independently on a particular model. Such approach, even if an object-oriented language is used, has several drawbacks. First of all the hadronic model commonalities as well as designing and programming experiences are badly exploited. For example, each new developer has usually started to develop a particular hadronic model from scratch. As a result of it, each new model developer starts from the analysis and design stages. The design duplications are manpower consuming. The different designs have different qualities and they provide different degrees of satisfaction for the user as well as the future model developers requirements. The design duplications lead also to the code duplications. The particular hadronic model code quality depends strongly on the coding experience of model developer or developers. It also becomes more difficult to learn, maintain and extend a set of hadronic models created by different developers as well as to connect them for collaborative work. As a rule, the hadronic model developers are not the software experts, they are physicists and experts in their subject domains. For them it can be difficult to find a proper solution of the specific software tasks.

Thus, we advocate the development of an object-oriented framework to take into account hadronic model commonalities, to facilitate user work and to increase productivity of the developer work. Within it a hadronic model developer can be more deeply concentrated on the particular model problems. He or she needs to write much less code since an essential part of the program already exists. He or she does not need to be a software expert to write robust code. A new model code inherited from the framework could also be much easily tested since it is already integrated with the rest of framework.

4. FRAMEWORK BASIC IDEAS

As was already discussed, the hadronic models as well as their usage and their development have a lot of commonalities. We tried to separate commonality from variability in the hadronic model interfaces and application logics during the framework development.

Here we begin to describe the first version of the hadronic MC model framework, which is suggested as the starting point. Let us explain the framework basic ideas and its collaboration with users and model developers.

We made an attempt to formalize hadronic model components as unit blocks to construct a composite hadronic model. All such blocks can be stored as an exendible collection of the model components. The definition and design of the unit component blocks offer a universal, flexible and powerful mechanism for a hadronic model construction. The question is how to define such components? Any model component can be structured into an interface part and the part presenting its application logic. The interface part of the model component allows component collaborations. With the help of this part a user can also handle model component

parameters, its input-output, its execution process, etc. Such formalization becomes visible, when we provide the component interface standards.

Particularly, from model components we can separate the components with the «runnable» interfaces, i.e.. they include methods to execute them and support user input. We call a model component as the main component if it uses other model components and supports the runnable interface.

On the other hand, the interface standards, if they are required, dictate a hadronic model developer to follow definite rules during a component implementation. These rules can be taken into account by means of the component frames, which are created for the command line user interface, or by means of the component wizards in the case of the graphical user interface. The component wizard helps us to create a particular case of a component. We call it the empty or skeleton component. A developer can also create a skeleton using the component frames and performing the required editions. Thus, the interface standards facilitate the developing of the model component interface part.

A hadronic model developer should mostly work on the implementation of a particular application logic. The part, which presents an application logic can also be universal. It is known that the same application logic can be used in several physical models. Below we will touch more details of the application logic universality.

In the case of developing a composite hadronic model there are very important questions about the component control, their collaboration and their collaboration with a user. Particularly, a component collaboration should allow substitution of variable implementation of different application logics (e.g., the alternative components) via a common interface. Again the interface standards help us to suggest mechanism for model component control and their collaboration. This mechanism is provided by the framework control subsystem. The control subsystem is implemented by means of two central concepts of the framework: the resource and the dispatcher. The resource contains static information about model components. The dispatcher is the heart of our framework. It loads components, obtains information about components, creates needed files, and so on. The dispatcher is the chief of all components. It controls all inner model processes at runtime. But any model component is controlled by the dispatcher in the same way by means of the set of standard messages. The dispatcher helps us to organize multithread work of our framework. The dispatcher concept allows an easy way to integrate our framework into the «more global» package, e.g., the *GEANT4* [1] or the *ROOT* [5].

Our framework is a tool to perform simulation of hadronic interactions. Therefore, we have suggested a uniform and powerful output subsystem with main goal to facilitate simulated data visualisation and analysis. This output subsystem is based on the extendible data transfer class library.

The data transfer class library as well as the application logics of the model components are closely connected with the problem domain. The objects of the data transfer classes are used for an information exchange between hadronic model algorithms. These classes help us to store the history of an event generation and to implement the universal model algorithms. Again the data transfer class library helps a hadronic model developer to be concentrated on the application logics development.

At the end of this chapter we want to say a few words about the object identifiers. Any object of our framework has its unique identifier *(ID)*. At the moment any *ID* can have the unsigned value, thus we have a possibility to assign more than 4 billion different values. The

unsigned values are very convenient for searching and navigation. The knowledge of object's *ID* helps us to obtain full information about this object. The object identifiers are heavily used in our framework subsystems. Particularly, we are developing the framework help subsystem based on the *HTML*. It is possible due to that the unique object identifiers can be binded with the *HTML* files describing objects.

5. POSSIBLE USER AND DEVELOPER SESSIONS

For communications with user we have developed the user interfaces. Here we explain the user-framework interaction mainly by means of the command lines. However, a user using the Windows platform can run the graphical user interface.

First of all a user is able to visualise the hadronic model component list and choose or register needed component. The *HTML* based help subsystem under development will offer a user needed model documentation. After model component registration a user can visualise and edit the default model parameters and the framework will provide parameter consistency check. The visualised set of the hadronic model parameters can have a tree structure with the leaves are groups of parameters. The hadronic model parameter set can have three states: default state, current state and the state, which includes one or several previous states stored in a file. Thus a user can either keep a default value of a parameter variable or change this value. Also a possibility for user to store current value of parameter variable on a file is offered. It can be done in order to give the «undo» possibility, i.e., a user will always be able to get back the old parameter values.

A hadronic model can be configured by its parameter edition. Even more advanced possibility for a user to configure a composite model is offered. He or she is able to substitute a model component by an alternative model component without changing the code (see the component substitution chapter).

Then a user can customize output. Similarly as a parameter set, the output set can be a tree, where leaves are channels and branches are groups of channels. Each channel or group of channels can be in the enabling or disabling states. By default they are forced to be in the disabling states. Using this scheme we offer a possibility for a user to control the output information. For example, preparing an output of a particle object we can obtain either full information about a particle, i.e., its momentum, its position, its encodings, its spin, its electric and baryon charges, etc., or only its momentum and its position. Also in the case of a multicomponent model to obtain the history of the generated event a user can activate any hadronic model component or all hadronic model components to write information about its work.

The input set is organized similarly as the parameter set since both the input data and the parameters require that their values should be checked. For each particular model the input maps (see an example of input maps in chapter 17) are offered and a user will be able to edit them and again the framework will check the input consistency.

For particular hadronic model a user is able to overload default aggregated component set, default parameter set, default output configuration. It is useful, e.g., in the case, when a hadronic model was tuned for the best description of experimental data and a user is interested only in generated final states. Such situation is typical if a model is used for the applied purposes to predict missing information about hadronic reactions.

The hadronic model run control is offered by the model runtime information: information messages, warning messages and error messages.

Since the framework has uniform output subsystem it helps us to create the generated data analysis and visualisation subsystem. Now simple data filters and some plot facilities are provided for the user, who uses the graphical user interface. We should note that within the graphical user interface the user-framework collaboration is similar as in the case of the command line user interface. It is going through the parameter, input, output, runtime information, etc., graphical windows. In this case a user deals with several threads, e.g., the run process or active model component thread and the data analysis thread. In the command line user interface, a user operates with the corresponding ASCII files.

Another possibility of performing more detailed analysis of simulated events is the use of an external analysis tools. The output subsystem is able to prepare the output data to be suitable as the input (macros) for the *ROOT* [5] analysis and visualise tool.

For efficient work a developer can learn the set of framework classes (see also chapters below): **NMAlgorithm** class as the starting point to implement different model components, several classes, such as the **NMDoubleParam**, the **NMIntParam**, etc., to handle model parameters and inputs as well as **NMHistory**, **NMBaseParticle** and **NMPrimerParticle** classes to extend the data transfer class library if it is needed. For other framework classes a developer can learn only their headers. It is also better if a developer makes the acquaintance with the hadronic application logic library to avoid a universal algorithm duplication. The implementations of some classes are closely connected with the interface system, e.g., command line or graphical user interface. Their implementation should be hidden from a model developer since we may want to change them without affecting the developer's codes.

To help a model developer we provide several model components frames as well component documentation frames. An example of such frame can be found in the appendix chapter. If a developer wants to implement a new model or a new model component, e.g., as an alternative model component, he or she can choose suitable model component frame, edit it according to the frame comments and implement required methods. The component documentation frames help a developer to create the help documentation about a component.

The simplest case of a developer's work is that a developer wants to implement foreign working code as an alternative model component. Particularly, to implement foreign runnable component code a developer has to use the runnable algorithm frame, where only two methods: *OnRegisterInputMap()* and *OnRun()* (see the appendix chapter) should be implemented. A developer has also to define the input map and map contained variables and register them by *OnRegisterInputMap()*.

6. MODEL COMPONENTS AND THEIR STRUCTURES

We distinguish different kinds of model components: the algorithm component, data analysis component, table component, etc. Each type of component can have its specific properties and methods. Such approach has some advantages for a model developer, since we can provide the standard frames for all components to facilitate component implementations. It is also rather flexible to allow the implementation of complicated model [6]. From the hadronic model developer point of view the most important model component is the algorithm

component. Let us explain the details of its structure. Below we will call it shortly the model component or the component.

For this component we separate the component factory and the component static information parts. We need the component factory part, because the creation of objects is very important question since we want to have a possibility for dynamic binding, i.e., substitution of a component by the alternative component without changing the code as well as object creation only, when it is needed. Another aspect of this question is connected with model component object collaboration (see below). There are two ways, when one object uses another object. In the first case we can declare it as a member and we cannot substitute it during run time. In the second case we declare a pointer to another object and we will get a runtime substitution. Thus we can substitute this object by its child or derived object. But we have to assign this pointer. As a rule it can be done by a set method or using a constructor. It is not always good because we may want to use a complex object and we need to know in aggregating object all information about aggregated object. To avoid such problem we can create the proxy object [3] of an object, which is planned to be used. It can be a pointer to the component identifier (*ID*). During construction time all proxy objects are gathering into a list. As a result, each object knows, which aggregated objects can be used. But how can we create these aggregated objects? It can be done by means of a factory method [3].

The component static information part or component resource part contains the information, which is needed to create this component. It also contains the pointers (references) to documentation. Besides the component factory and its static information parts, a component can include methods and data for model parameters, model input maps and model output configurations as well as the execution or run methods.

To provide more flexibility for a developer, we also classify the algorithm components according to their interfaces: the so-called «runnable» components, which include the execution and input methods. Such components can be used as the separate models or as the main model components, which include aggregated components. We distinguish also the «general» components, which can be used only as the aggregated components. We should note that any runnable component can be used as a general component. We classify the «virtual» components, i.e., the «virtual runnable» or «virtual general» components, which are similar to the «runnable» and «general» components. The difference between them is in the component factories. It is not possible to create the virtual component objects.

At the end of this chapter we would note that the **NMAlgorithm** class is the base class to derive the algorithm component classes (see the appendix chapter), which should be implemented. Particularly, it supports three methods: *OnOverloadDefaultProxy()* to overload the default component, the *OnOverloadDefaultParam()* to overload default parameters and *OnOverloadDefaultOutputConfig()* to overload default output configuration.

7. FRAMEWORK CONTROL SUBSYSTEM

Let us imagine that all needed model components are implemented. How is the user's interaction with the framework libraries provided? How will components collaborate? As we have already mentioned, all interactions between the framework libraries and user interface system as well as model component collaborations are provided by the framework control subsystem. It is hidden from a user. We denote such control subsystem as the dispatcher.

Thus, the dispatcher is a shell between the user interface and the framework libraries. For the command line interface, the framework control subsystem is currently implemented as the **NMDispatcher** class, which keeps the pointer to the model component base class as well as the vector of the model component factories. It includes many methods (see chapter 17) to support several phases of the user-framework interaction: creation phase, edition phase, execution phase and destruction phase. Particularly, it allows one to obtain full list of the model components included into the framework. The dispatcher object can load, run and delete a model component. The dispatcher object checks a possibility of substituting a component by an alternative component. The dispatcher object also manages parameter and input writing and reading. It manages writing and reading of the output configurations, etc. There are two groups of methods in the dispatcher. The first group consists from the methods needed to receive the information about framework library contents and to control this information, e.g., the method to obtain the component tree from the global list. The second group are the methods, which control currently loaded components. The dispatcher methods are named according to the rule: *Action + ComponentType()*, e.g., *LoadAgorithm()*, *LoadAnalyzer()*, *LoadTable()*, *DeleteAlgorithm()*, etc., since we have different model components.

8. PARAMETERS AND INPUT SUBSYSTEMS

Our framework has the set of classes to support model parameter and model input handlings. A developer to implement the parameters of his model component has to define parameter type, assign its default value and a comment for it. Provided model component frames facilitate this task for a developer. After model component development a model user will get a possibility to edit parameters under the framework control. A developer can also bound parameter values by means of some parameter method overloading.

To solve the task of the input data set we suggest the input maps based on the lists of simple data types. By means of the input maps we organize not only the uniform input for a model user, but we also save developer time since our framework performs needed input map data converting.

The **NMParam** class initiates the parameter/input class branch. This class keeps common properties of parameters, e.g., parameter names, methods to write parameters into data file, to read parameters for their visualization, to check parameter consistency, to reset parameters, etc. The **NMParam** class gives rise to the **NMParam** group of classes: **HM***Param** and **NMParamGroup**. The **NMParam** realizes common features of a simple parameter. The necessity of such a class is connected with difference between writing into file for a simple parameter and for a group of parameters independent from parameter type. The **NM***Param** presents parameters of different types. The stars in the parameter class name indicate that any name can be used according to the rule: **Bool + NM***Param = NMBoolParam**. The **NMParamGroup** class allows us to join parameters into named group.

As we already mentioned the parameters are structured like tree and have different states. A model component can contain only one group of parameters. Thus, the parameters tree is the component tree, where leaves are groups of parameters. The parameter tree is created by the proxy component tree. There are default state and current state. Also several former states can be stored in a configuration file. If the configuration file is not empty, the last state of a parameter tree will be loaded from this file.

Similar structure is also suitable for model input. To support input maps we use the **NMInputMap** class, which is a redefinition of the **NMParameterGroup** class.

9. MODEL COMPONENT SUBSTITUTION

Our framework approach allows substitution of model components by the alternative components without changing code. It gives a user the possibility to change hadronic model structures. Two types of the substitution are possible. The first one we call the static substitution. Let us image the next situation. There are several model components that have been developed. We denote them as *A*, *B*, *C* and *C1*. The component *A* uses the component *B* and the component *B* uses the component *C*. When a developer creates these components, he or she knows nothing about the component *C1*, which is an alternative for the component *C*. A user applying the method *OnOverloadDefaultProxy()* in the component *A* can force the component *B* to use the component *C1*. The implementation of the *OnOverloadDefaultProxy()* method can look as follows:

```
ClassA::OnOverloadDefaultProxy()
    {
    OverloadProxy(ID_C1, ID_B, 0)
    }
```

The dynamic substitution can be performed at run stage. For example, a user would like to use the component *C* instead of the component *C1*. He or she is able to solve this task by means of our framework. Within the command line interface version it can be done by edition of the parameter file, which has the *.nmp* extension (see also chapter 17). The parameter file edition includes the replacement of the lines related to the component *C1* by the lines related to the component *C*. From the C++ programmer point of view such substitution is provided by the subclassing mechanism, i.e., alternative components need to have a common base class. The developed component frames of our framework supports such subclassing.

10. DATA TRANSFER CLASS LIBRARY

The data transfer classes organize the data transfer between model components. This is the first goal to develop such library. It helps us to store the history of an event generation. Any object of the data transfer class supports serialisation, i.e., it has methods to read and write its state. The data transfer classes use the exchange data format described below. Thus, the data transfer class library helps us to create uniform and powerful output subsystem to obtain the result of simulation. For this purpose any data transfer class is derived from the **NMHistory** class with the aim to obtain an output of the generated event history. It has two methods *Config()* and *Out()* that are implemented to write their objects states. (See examples of the method implementations in the framework output subsystem chapter). This is the second motivation to create such hierarchial library.

We would stress that the presence of these classes allows a developer to pay more attention to the algorithmic logic part of a hadronic model development, because a developer need not think about details of the input-output operations.

A navigation system can be offered (we are working on it) and a developer can navigate through the data transfer classes. He or she can visualise any such class and apply the data transfer class wizard with the aim to extend the data transfer class library.

The data transfer classes are tightly connected with the hadronic models domain. The data transfer class hierarchy represents physical objects from this application area. They describe three groups of such objects. The first one can be called the simplest objects or the objects without inner structure. Then we can separate collections of the simplest objects. Describing the simplest object we do not need the information about its inner structure. We have developed the **NMBaseParticle** class. It has two members: the 4-momentum and the 4-position, and represents the relativistic phase space point object or relativistically moving point object. This class can be considered as an example of the data transfer class, which represents the simplest object. Other examples of such classes, which are derived from the **NMBaseParticle** class, are the **NMParticle** class, representing the elementary particles, the **NMParton** class, representing the quarks and gluons, and the **NMSimpleNucleus** class, representing a nucleus without its nucleon structure. The simplest objects can be combined into the different object collections. The members of a collection do not have structural dependences. For example, object collections can be created using the **NMBaseParticleVect**, the **NMParticleVect**, the **NMPartonVect** and the **NMSimpleNucleusVect** classes. They keep arrays of the pointers to the simplest objects. The composite objects need the information about their internal structure. The classes, representing such objects, have the collection classes as the members or they are derived from the collection classes. As the examples of this group of classes we can consider the **NMNucleus** class, which has as a member the **NMParticleVect** class, and the **NMExitedString** class, which has the **NMPartonVect** class as a member.

For many applications of any class hierarchy it is very convenient to have a base class, which joins all classes. As a rule, it is pure virtual class. It factorizes the common properties of all objects of this hierarchy. The relativistic phase space point object is commonly used in the high energy hadronic MC models. In our data transfer class hierarchy we use for that the **NMPrimeParticle** class, which represents the most operations needed to work with the relativistic phase space point. This class gives the rise to all other classes of the hierarchy.

Our data transfer classes represent not only the dynamical properties of the physical objects (momenta, positions, etc.), but also their static properties (charges, encodings, etc.). For example, the **NMParticle** object describes the dynamic properties of the elementary particle and has a pointer to the static object, which keeps the information about mass, decay branchings spin, etc. The static information is stored as tables. We have other tables, e.g., the interaction cross section table. We are working on the service tools (framework database service tools) to support tables within our framework.

At the end of this chapter we want to mention another important motivation to create the hierarchial library of the data transfer classes. It is connected with the algorithm logic universality. The degree of universality of an algorithm is determined by the amount of needed input-output information. Our data transfer classes have an hierarchial structure, the most universal algorithms are those, which use as input and output the objects of the node data transfer class.

11. FRAMEWORK OUTPUT SUBSYSTEM

The output phase is the most complicated phase of interaction between the framework and a user. Let us make several things to be explained before description of the output phase.

There are different approaches to write and read data. One of them is that an object writes and reads data by itself. In this case we need this object to be created before reading. The configuration to be read is provided by this object class and we need not store the configuration on a file. But in such a case the data stored on a file cannot be read from outside of a package, which includes the object class. We have decided to use another approach. We write the configuration to be read at the beginning of a file. This configuration is based on the list of basic types. These types can be read within any other package.

We have developed the framework input-output file system to avoid dependence on the different platforms. Particularly, it allows us to open and close a file, to read data from a file and to write data on a file.

Let us consider the output data structure, which can be written on an output file. It includes the output file header, an extra information, the channel definition, the event configuration and the data part.

The output file header is needed to identify the output file. It contains the identifier field, the file version and some offsets to the other parts of file as the channel definition, the event configuration and the data. An extra information, which can be used by a developer for some reasons, will be placed just after the header.

We need a universal data exchange mechanism, which is independent of particular language or package. For this purpose we suggest a format for data reading and writing. A variable is stored in a channel with its own identifier. With each channel identifier we bind the channel type and some extra information to organize convenient user interface. Several channels can be grouped into a group of channels with unique group identifier. Each identifier of channel or group of channels can have prefix *REP*. It signals that given channel or given group of channels is repeatedly stored. To describe a channel we use four fields: *ChannelID*, *FullName*, *ShortName*, and *UnitName*. The *ChannelID* field keeps information about the channel type and it can also be used in the help subsystem. The channel full name *FullName* field is used by the user interface. The *ShortName* field can simply keep the short name of a channel. For example, the *X* component of the Lorentz vector momentum can be presented as follows: 1003, «X-component of Lorentz vector», «px», «GeV». The *ShortName* field was also introduced to be a variable name of this channel. Using this field we can customize the output data structure with the aim to produce output, which can be accepted as an input for the foreign package. For example, it helps us to create the *ROOT* [5] macros. The last field *UnitName* informs a user about the physical units of channel variable values. To support channels we have developed the **NMChannelDef** class and its vector **NMChannelDefVect** class.

We have introduced program event objects. We define a program event object as an unit of information to be written. To support it we have developed the **NMProgramEvent** class. Each program event object has its own *EventID* and can have its name to be used in the user interface. In order to read the binary data from the output file we have to know their configuration. The configuration can be found by means of the *EventID* in the configuration part of the file. From this point of view the *EventID* can be considered as an identifier of the event object type.

The configuration is array of *ChannelID*, which are included in this event. Each model algorithm component supports the method *OnRegisterOutputConfiguration()*. We use this method to register configurations of program events, which can be written by this component. The registration procedure is very easy. It adds the channel identifiers or group of channel identifiers to the array. An example of the method *OnRegisterOutputConfiguration()* implementation can be found in chapter 17.

We said already that the data exchange between the model components is supported by the data transfer classes. These classes play an important role in the output subsystem. In order to solve the task of writing event configuration to the output file all these classes have to realize the method *Config()*. We present below two examples of this method implementations.

```
void NMBaseParticle::Config(NMProgramEvent& ProgramEvent, bool bHistory)
    {
    ProgramEvent.AddGroup(bHistory, 10, true, "Moving point",
"MovePoint");
                ProgramEvent.AddDouble(101, true, "X-position", "X",
"fermi");
                ProgramEvent.AddDouble(102, true, "Y-position", "Y",
"fermi");
                ProgramEvent.AddDouble(103, true, "Z-position", "Z",
"fermi");
                ProgramEvent.AddDouble(104, true, "T-position", "T",
"fermi");
                ProgramEvent.AddDouble(105, true, "X - component of
Lorentz momentum", "PX", "GeV");

                ProgramEvent.AddDouble(106, true, "Y - component of
Lorentz momentum", "PY", "GeV");
                ProgramEvent.AddDouble(107, true, "Z - component of
Lorentz momentum", "PZ", "GeV");
                ProgramEvent.AddDouble(108, true, "Energy", "E", "GeV");
    ProgramEvent.AddEndGroup();
    }


void NMParticle::Config(NMProgramEvent& ProgramEvent, bool bHistory)
    {
    ProgramEvent.AddGroup(bHistory, 30, true, "Particle");
                NMBaseParticle::Config(ProgramEvent);
                ProgramEvent.AddDWord(301, true, "Encoding", "Enc",
"Encoding");
                ProgramEvent.AddDouble(302, true, "Mass", "M", "GeV");
                ProgramEvent.AddDouble(303, true, "Charge", "M");
    ProgramEvent.AddEndGroup();
    }
```

The data transfer class objects do not exist, when we create the output configuration, as a result, these methods have to be the static methods.

We define event configurations before run session. It gives us a possibility to control the output. Particularly, we are able to protect any channel or group of channels from their output. By default all channels are open and all program events are closed for their output except the program events from the main model component.

The framework supports also two predefined event objects, which allow us to write to output the values of input map and parameter of model components, which are executed in the current run session. For them the registration of channels is performed automatically. However, a user is able to mask these events or some their channels with the aim to forbid their output.

There are different *Out()* methods, which are responsible to write the data part of the output file. We should note that before to write data on the output file each such method checks that there are channels, which open for the output. As we already mentioned each data transfer class has such method. For example, for the **NMBaseParticle** and the **NMParticle** it has been implemented as follows:

```
void NMBaseParticle::Out(NMOutputConfig& Config, NMFile& File)
    {
    BEGIN_OUT_GROUP
        {
        OUT(X());
        OUT(Y());
        OUT(Z());
        OUT(T());
        OUT(PX());
        OUT(PY());
        OUT(PZ());
        OUT(E());
        }
        END_OUT_GROUP;
    }


void NMParticle::Out(NMOutputConfig& Config, NMFile& File)
    {
    BEGIN_OUT_GROUP
        {
        NMBaseParticle::Out(Config, File);
                OUT(Encoding());
        OUT(Mass());
        OUT(Charge());
        }
    END_OUT_GROUP;
    }
```

In the above *Out()* method example the *OUT(Variable)* is the macro to write a *Variable*.

Let us say a few words about the output of the generated physical event history organization using as an example the **NMProgramEvent** class. It has also the *Out()* method. The *Out()* method of **NMProgramEvent** class writes on output file the header (*EventID*, *EventCounter*, and *HistoryCounter*) and then binary data. Each call of the *Out()* method can be considered as the event object creation and it can be identified using the *EventCounter*. There will be no event objects written on the output file with the same value of the *EventCounter*. The *HistoryCounter* field is used to make connection between event objects and model component objects, which are responsible for the event object creation. The *HistoryCounter* variable is the only a member of the **NMHistory** class. With the aim to obtain the output of the generated physical event history, we have also derived all data transfer classes from the **NMHistory** class. But in some cases an event object can contain not only objects of the data transfer classes. There will also be event objects, which do not contain any object of the data transfer classes. If so, we can also write within events the values of the *HistoryCounter* as a member.

At the end of this chapter we want to discuss the generated data analysis procedure, which can be performed within the graphical user interface of our framework. We consider this procedure as the three-steps procedure. At the first step one can perform the structure filtering since the data are organized as a tree, where nodes are enumerated by their *ID*. It gives us a possibility to select data by using only the data structure information, which is stored at the beginning of data file. The next step is to perform the data sensitive filtering, i.e., one can select data using themselves. The last step is that selected data can be stored and presented as the one-dimensional histogram and two-dimensional plots.

12. RUNTIME INFORMATION

Our framework supports the runtime information output. By means of it the model components inform about their execution processes during run session. The runtime information is simple text based information. There are three types of the runtime messages: information messages, warning messages, and error messages. The information messages are used to tell about normal execution process. The warning messages inform a user about potential errors or other situations, which are able to destroy normal execution process. The execution proccess will be continued after the appearance of the warning messages. The error messages appear, when the execution process will be aborted. The error messages inform also a user about place and type of the error. The framework detects itself the information about place of an error and a developer need not make special efforts to solve this task.

A user has a possibility to control the runtime information output. Particularly, he or she can either totaly suppress the information messages or suppress the information messages from some model component objects. The warning messages are divided into some groups. A user can suppress any group to be appeared in the output or overload any group to produce error messages. Each warning or error message has also its own unique *WarningID* or *ErrorID* identifier. It allows us to help a user or a developer to obtain the full information about the reason of warning or error.

Let us provide more details about the possibility of obtaining the runtime information. We have developed the **NMOwner** component base class, which includes the methods: *Message()*,

Warning(), and *Error()* to support runtime messages. The error method calls also exception to close the execution process. All objects of the model component classes have pointers to their context objects. The work to organize the runtime output is performed by the context object of a model component. The dispatcher creates the context objects and informs them, where the runtime information has to be placed.

13. FRAMEWORK HELP AND DOCUMENTATION

The help subsystem based on the *HTML* is under development as well as printed version of the framework manual [6]. Such subsystem is needed to help a user and a potential hadronic model developer. They can obtain parameter description, input description, physics description, numerical algorithm description, etc. The help subsystem from a hadronic model developer point of view can be considered as a platform to explain how to develop a model code.

As we already said, any object of our framework such as component, parameter, error message, etc., has the unique *ID*. It allows us to bind these *ID* with the object describing *HTML* files. For version with the graphical user interface we will provide some tools to work with the help information, e.g., there is a possibility of obtaining the context of help information. This help subsystem will be developed with the aim to provide either convenient access for help information or uniform way to add new help information by means of typical documentation frames. The user of help subsystem need not know the details of its work in order to find or add help information. The component documentation frames allow one to present information for the standard appearance. These documentation frames are different for different types of model components. They are developed in the complement of model component frames. A model component developer must follow the instructions, which can be found in the documentation frames. There are also classes such as the **NMDeveloper** and the **NMDeveloperList**, which organize the personal information about model developers, and the **NMWEBDocument** and the **NMWEBDocumentList** classes, which assist to manage the model documentation on the *Web*. See also the example of runnable component frame in chapter 17.

14. COMPONENT LIBRARY

Discussing different aspects of the hadronic model framework, we have paid attention mostly to its interfaces. But the real hadronic model component to be used for the reaction simulation should have its application logic. The hadronic model algorithm transforms an input into an output. The objects of the data transfer classes are served as the input and output objects for the model component objects. In the runnable model component frame the *OnRun()* method name was chosen as the standard name for the transformation method. Thus, we assume that the implementation of this method will be the main job of a hadronic model developer. It usually requires hard developer's work and needs not only the software testing, but also the physical testing. The large number of the hadronic model runnable components, e.g., such as the **NMElasticScatterer** and the **NMPomeronPartonStringModel** (see for model physics description [4]) as well as hadronic model virtual and general components have been

already implemented and included into the component library. The component library will be described elsewhere [6].

15. UTILITY CLASS LIBRARY

The utility classes contain useful data and methods: random number generators, different integrators, equation solvers, physical units and constants, etc., which are often used. These classes accumulate the working experiences of the hadronic model code programmers. Some of such utility classes can also be found in the *CLHEP* library [7]. Very convenient strategy to use physical units and constants was adopted from the *GEANT4* [1].

16. CONCLUSIONS

We have shortly discussed the first version of the object-oriented framework for the hadronic MC event generators. We made an attempt to explain that our framework has different features to facilitate the hadronic model user and developer works. As the framework name we suggest to use the *NiMax* word.

In conclusions, we would only stress the main point of our approach to develop the hadronic MC event generators. It is the component approach, when a complex hadronic model is composed of small and simple pieces, that are self-contained entities (see, e.g., the book [8]). The component approach has many advantages and we can enumerate several of them. It allows us to formalize the particular type of components by separating interface part and application logic. We have offered different component frames for hadronic model developers. Thus, a model developer should work on the component application logics and each component application logic can be developed independently of other component application logics. Model components can be composed in a variety of ways and the new components with their peculiarities can be added, that offer a flexibility for the construction of a powerful hadronic model. The different implementations of a component application logic can be interchanged at runtime enabling a hadronic model user to obtain the needed model properties without redesigning a model and writing the model code.

Working on the framework we have obtained good software experience and we understand that first framework version should be more tested, improved and extended to be the successful tool for its user and consider it as the starting point. We hope also that our experience to develop the hadronic model framework as well as the framework ideas can be applied not only in the hadronic model application domain, but can be suitable to develop other similar applications.

Finally, we would like to thank the members of the GEANT4 hadronic group for the discussions with respect to the hadronic MC generator design. One of us (N. Amelin) is grateful for S. Giani, who invited him to work on the hadronic models for the *GEANT4* toolkit. Our MC generator framework idea was born during this work. M. Komogorov would like to express special thanks to W. Trzaska for collaboration during his work at Physics Department of the Jyväskylä University.

17. FRAMEWORK WORKING EXAMPLES

With the aim to illustrate possible framework user and developer sessions we show several examples: The first is an example of the framework main function, which can be applied in the case of the command line user interface. This example explains the dispatcher importance. Then comes the example of the runnable algorithm component frame illustrating a user and a developer works with the parameters, the input maps and the output configurations. We show also the example, when a user is able to substitute a model component by the alternative model component. Finally, we would demonstrate an example of the heavy ion collision simulation performed within the Pomeron based Parton String Model (see the model physics description [4]).

17.1. Dispatcher at Work. Let us look at the first example. As a result of this main function execution a user can obtain the list of the model components (see file «Content.hm») as well as the information about registered (the model *ID* has been assigned) model parameters («.nmp» file), model input maps («.nmi» file) and model output configuration («.nmo» file). The corresponding files can be edited. Model user can execute registered model and obtain the configured output, i.e., the result of the hadronic reaction simulation, which will be written on an output file («.nmd»). In this example of the main function one can see the key role of the dispatcher. It is doing all control job, which is hidden from a user.

```
#include <iostream.h>
#include "NM.h"
#define MAX_BUFF 256

void main(int cArg, char* aArg[])
{
    Dispatcher.RegisterFactory();
if (cArg < 2)
{
        cout << "To get help, please, use:\n"
"<NiMax -I>  - list of components\n"
"<NiMax -I ID [-iMapFName] [-pParamFName] [-oOutFName]>\n"
"  Subkey:\n"
"  -i - Input maps\n"
"  -p - Parameter list\n"
"  -o - Output configuration tree\n"
"To run a component, please, use:\n"
"<NiMax -R ID [-iMapFName] [-pParamFName] [-oOutFName][DataFileName]>\n"
"  -i - Input map file\n"
"  -p - Parameter set file\n"
"  -o - Output configuration tree file\n"
" DataFileName  - Output file name\n";
        return;
        }
if (strcmp(aArg[1], "-I") == 0)
{
```

```
if (cArg < 3)
{
Dispatcher.OutComponentsList("Content.nm");
return;
}
int ID = atoi(aArg[2]);
if (!Dispatcher.LoadModel(ID))
    {
     cout<<"There is no component with ID = "<< ID << endl;
     return;
    }
if (cArg < 4)
{
char Buff[MAX_BUFF];
itoa(ID, Buff, 10);
strcat(Buff, ".nmi");
            Dispatcher.InputToTextStream(Buff);

itoa(ID, Buff, 10);
strcat(Buff, ".nmp");
            Dispatcher.ParamToTextStream(Buff);

itoa(ID, Buff, 10);
strcat(Buff, ".nmo");
            Dispatcher.OutputConfigToTextStream(Buff);
return;
}
for(int c1 = 3; c1 < cArg && c1 < 6; c1++)
    {
     if (strnicmp(aArg[c1], "-i", 2) == 0)
 Dispatcher.InputToTextStream(aArg[c1] + 2);
     if (strnicmp(aArg[c1], "-p", 2) == 0)
 Dispatcher.ParamToTextStream(aArg[c1] + 2);
     if (strnicmp(aArg[c1], "-o", 2) == 0)
 Dispatcher.OutputConfigToTextStream(aArg[c1] + 2);
}
}
if (strcmp(aArg[1], "-R") == 0)
{
if (cArg < 3)
{
Dispatcher.OutComponentsList("Content.hm");
            cout<<"\nPlease! Set the ID\n";
return;
}
int ID = atoi(aArg[2]);
```

```
if (!Dispatcher.LoadModel(ID))
{
cout<<"There is no component with ID = "<< ID << endl;
return;
}
for(int c1 = 3; c1 < cArg && c1 < 6; c1++)
    {
     if (strnicmp(aArg[c1], "-i", 2) == 0)
 Dispatcher.InputFromTextStream(aArg[c1] + 2);
     if (strnicmp(aArg[c1], "-p", 2) == 0)
 Dispatcher.ParamFromTextStream(aArg[c1] + 2);
     if (strnicmp(aArg[c1], "-o", 2) == 0)
 Dispatcher.OutputConfigFromTextStream(aArg[c1] + 2);
}
        Dispatcher.Run("kyky.nmd");
        }
}
```

17.2. Runnable Model Component Frame. As we already said, a model developer to create a model component can start the component implementation by editing the suitable model component frame. For example, the runnable algorithm component can be used. Its header and implementation are shown below. We shall assume that all members are public to simplify the discussion; the extension to protected and private is not straightforward. We plan to explain it in the extended description [6] of our framework. Let us at first explain the parameter, input and output parts of the runnable model component. Below we consider implemented parts of runnable component for the hadronic string decay model. This model performs string decay simulation, i.e., it models the process, when a string with the parton ends and its mass decays into stable and resonance hadrons (see the model physics description in [4]).

17.2.1. How to Define, Edit and Check the Model Component Parameters? In order to define parameter one has to declare the member of component class. There are corresponding commented lines in the runnable component frame (see below). Let us consider a part of the **NMStringDecayer** model component header. Within it we declare two double parameters: *DiquarkSuppression* and *ClusterMass*.

```
class NMStringDecayer : public NMAlgorithm
    {
public:
    DEFAULT_LOCAL_DEFINITION(NMStringDecayer, NMAlgorithm);

public:
    // Messages

public:
    // Proxy
```

```
public:
    // Parameters
    NMDoubleParam ClusterMass;
    NMDoubleParam  DiquarkSuppression;
public:
    // Input

public:
    // ProgramEvent

public:
    // Methods

};
```

The framework is able to control these parameters, if they are registered by the *OnRegisterParam()* method. Below we show an example of the method implementation.

```
void NMStringDecayer::OnRegisterParam()
    {
     Add(ClusterMass.Init(200908, "Cluster mass", 0.15, GeV,  "GeV"));
     Add(StrangeSuppress.Init(200910, "Strangeness suppression", 0.44));
    }
```

Within the command line user interface the component parameters can be written on the file with the extension *.nmp* and read from it. Below we show the content of *.nmp* file for the registered parameters from the above examples.

```
"NMStringDecayer", 2009
        {
        "Cluster mass", "GeV", 200908,            0.15;
        "Strangeness suppression", "", 200910,   0.44;
        }
```

We can edit this file, e.g.

```
"NMStringDecayer", 2009
        {
        "Cluster mass", "GeV", 200908,          0.18;
        "Strangeness suppression", "", 200910,  0.40;

        }
```

This file can be used as the parameter of the framework execution file. Let us call it as the *NiMax.exe*. Also framework gives a possibility to check edited parameter consistency. For this purpose a developer can overload the *OnCheck()* method, e.g.,

```
bool NMDoubleMinMaxParam:: OnCheck()
{
 if (Value < Min || Value > Max)
 {
Warning(123,ERR_GROUP,"Parameter value is out of range. Default value
will be used");
  return false;
 }
  return true;
}
```

17.2.2. How to Define, Edit and Check the Model Component Input Maps? The definition of an input map is similar to the parameter definition, besides that, one has at first to declare input map. One can find the corresponding comment about it in the example of the runnable component frame (see below). Let us consider an example of the input map *InputMap1* declaration for the string decay component. This map has three fields: *LeftQuarkEncoding*, *RightQuarkEncoding*, and *StringMass*.

```
class NMStringDecayer : public NMAlgorithm
    {
public:
    DEFAULT_LOCAL_DEFINITION(NMStringDecayer, NMAlgorithm);

public:
    // Messages

public:
    // Proxy

public:
    // Parameters
    NMDoubleParam ClusterMass;
    NMDoubleParam  DiquarkSuppression;

public:
    // Input
    NMInputMap InputMap1;

    NMIntParam   LeftQuarkEnc;
    NMIntParam   RightQuarkEnc;
    NMDoubleParam   StringMass;

public:
    // Methods

};
```

Similar to the parameters, an input map and its content have to be registered. There is the method *OnRegisteredInputMap()* for that. Below we show an example of this method.

```
NMStringDecayer:: OnRegisterInputMaps()
 {
  NMAlgorithm:: OnRegisterInputMaps();
  AddMap(InputMap1.Init(2009100, "Input map 1 for string decayer "));
  InputMap1.Add(LeftQuarkEnc.Init(2009101,"Left quark encoding", 2,
"Encoding"));
  InputMap1.Add(RightQuarkEnc.Init(2009102,"Right quark encoding", 2201,
"Encoding"));
  InputMap1.Add(StringMass.Init(2009103,"String mass", 200, GeV, "GeV"));
 }
```

In command line user interface version this map can be written on the file with the extension *.nmi* and read from it. Let us look at the content of the *.nmi* file as a result of the input map registration, which was considered above.

```
"Input map 1 for string decayer ", 2009100
    {
    " Left quark encoding ", "Encoding", 2009101, 2;
    " Left quark encoding ", "Encoding", 2009102, 2212;
    " String mass ", "GeV", 2009103, 200;
    }
```

It is possible to offer different input maps for one component. For example, we can add the string momentum for string decay. To do that we have to add the declaration of new map and new input map values.

When somebody uses the input maps, then a map from the set of maps should be registered as the default map by the *OnRegisterInputMap()* method. If it is not done, then the framework will consider the first registered input map as the default one.

There can be some dependences between input or parameter values. For example, on string ends should be quarks with the opposite colours. To solve this problem, each runnable component supports methods *OnCheckParam()* and *OnCheckInput()*. An example of the *OnCheckInput()* implementation is shown below.

```
NMStringDecayer:: OnCheckInput(NM_ID ID)
    {
    if (IsColor(LeftQuarkEnc) && IsAntiColor(Right QuarkEnc) ||
        IsAntiColor(LeftQuarkEnc) && IsColor(Right QuarkEnc))
            return NMAlgorithm:: OnCheckInput(ID);
    Warning(127, ERR_GROUP, "String end quarks have the same colours");
    return false;
    }
```

17.2.3. How to Define and Edit the Output Configurations? We have already discussed that the output is performed by means of program events. Each event has to be declared. There is appropriate comment in the runnable component frame.

In the example, which is shown below, we add two program events for the **NMStringDecayer** component named *InitialState* and *FinalState*.

```
class NMStringDecayer : public NMAlgorithm
    {
public:
    DEFAULT_LOCAL_DEFINITION(NMStringDecayer, NMAlgorithm);

public:
    // Messages

public:
    // Proxy

public:
    // Parameters
    NMDoubleParam ClusterMass;
    NMDoubleParam  DiquarkSuppression;
public:
     // Output
    // ProgramEvent

    NMProgramEvent  InitialState;
    NMProgramEvent  FinalState;

public:
    // Input
    NMInputMap InputMap1;

    NMIntParam   LeftQuarkEnc;
    NMIntParam   RightQuarkEnc;
    NMDoubleParam StringMass;

public:
    // Methods

};
```

After declaration of the program events, they have to be registered. Within the command line version of the user interface the registered output configuration will appear in the *.nmo* files. Below we show the part of this file for the **NMStringDecayer** example.

```
"NMStringDecayer", 2009
    {
    "Initial state", 20090001, Off
        {
        "Particle", 20, On
```

```
            {
            "Moving particle", "MovingParticle", 10, On
                {
                "X-position", "X", "fermi", 101, On;
                "Y-position", "Y", "fermi", 102, On;
                "Z-position", "Z", "fermi", 103, On;
                "T-position", "T", "fermi", 104, On;
                "X - component of Lorentz momentum", "PX", "GeV", 105, On;
                "Y - component of Lorentz momentum", "PY", "GeV", 106, On;
                "Z - component of Lorentz momentum", "PZ", "GeV", 107, On;
                "Energy", "E", "GeV", 108, On;
                }
            "Encoding", "Enc", "Encoding", 301, On;
            }
        "Particle", 20, On
            {
            "Moving particle", "MovingParticle", 10, On
                {
                "X-position", "X", "fermi", 101, On;
                "Y-position", "Y", "fermi", 102, On;
                "Z-position", "Z", "fermi", 103, On;
                "T-position", "T", "fermi", 104, On;
                "X - component of Lorentz momentum", "PX", "GeV", 105, On;
                "Y - component of Lorentz momentum", "PY", "GeV", 106, On;
                "Z - component of Lorentz momentum", "PZ", "GeV", 107, On;
                "Energy", "E", "GeV", 108, On;
                }
            "Encoding", "Enc", "Encoding", 301, On;
            }
"String Mass", "Ecms", "GeV", 2009555, On;
        }
    "Final state", 20090002, On
        {
        "Particle", 20, On
            {
            "Moving particle", "MovingParticle", 10, On
                {
                "X-position", "X", "fermi", 101, On;
                "Y-position", "Y", "fermi", 102, On;
                "Z-position", "Z", "fermi", 103, On;
                "T-position", "T", "fermi", 104, On;
                "X - component of Lorentz momentum", "PX", "GeV", 105, On;
                "Y - component of Lorentz momentum", "PY", "GeV", 106, On;
                "Z - component of Lorentz momentum", "PZ", "GeV", 107, On;
                "Energy", "E", "GeV", 108, On;
                }
```

```
            "Encoding", "Enc", "Encoding", 301, On;
            }
        }
    }
```

Using the *On/Off* words one can open or close a channel to write it on the output file.

Finally, at the end of this subchapter we show an example of the runnable model component.

```
#ifndef _ClassName_
#define _ClassName_ 1

#include <NM.h>
//#include "ClassParentName.h" // Edit this line, if you use subclassing.

// Here, include the header files for each used child component.
//#include "ChildComponentName.h"

DEFAULT_GLOBAL_DEFINITION(ClassName, ClassParentName)

class ClassName : public ClassParentName
    {
public:
    DEFAULT_LOCAL_DEFINITION(ClassName, ClassParentName);

public:
    // Messages
    virtual void OnOverloadDefaultProxy();
    virtual void OnCreate();
    virtual void OnDestroy();
    virtual void OnRegisterParam();
    virtual void OnOverloadDefaultParam();
    virtual void OnRegisterOutputConfig();
    virtual void OnOverloadDefaultOutputConfig();

    virtual void OnRegisterInputMaps();
    virtual bool OnRun(unsigned, NM_ID);

public:
    // Proxy

    //NMAlgorithmProxy < ChildComponentClass > ChildComponentVariable;
    // Repeat the previous line for each used child component.

public:
    // Parameters
```

```
    // Here, define the parameters.
    NMDoubleParam P1;

public:
     //Output
    // ProgramEvent

    //Here, define the program events.
    NMProgramEvent Event1;

public:
    // Input

    //Here, define the input maps.
NMInputMap Map1;

NMIntParam     IP11;
NMIntParam     IP12;
NMDoubleParam  IP13;

NMInputMap Map2;

NMIntParam     IP21;
NMIntParam     IP22;

public:
    // Methods

    // Your methods and data members.
    };

#endif

#include "ClassName.h"


// Here, define name of your component and its own ID.
BEGIN_RUNNABLE_IMPLEMENTATION(ClassName, ClassParentName, ID , "ClassName")
    {
      // If you want to provide the full information about yourself,
      // please, use this structure.
   DEVELOPER("Physics design",//Explain your participation in the compo-
nent development.
     "FirstName SecondName Surname", // Your full name
     "Nickmame",                     // Your nickname
     "kyky1@cern.ch",                // Your e-mail
```

```
      "www.cern.ch",                    // Your WEB home page
      "7-096-216-53-18",                // Your phone number
      "7-096-216-53-18");               // Your FAX number

      // If you do not want to provide the full information about yourself,
      // please, use this structure.
Developer.Comment("Software design");
Developer.Name("FirstName SecondName Surname");
Developer.NickName("NickName");
Developer.e_mail("kyky1@cern.ch");
Developer.HomePage("www.cern.ch");
Developer.Phone("7-096-216-53-18");
Developer.FAX("7-096-216-53-18");
Add(Developer);
    // Repeat previous lines for each developer.

// Here, define the HTML pages, where your component is described.
      WEBPAGE("\\MyComponent\\Documentation\\File1.html",
//Path for the HTML file.
    "Physics reference");           // Explain the HTML file content.

      WEBPAGE("\\MyComponent\\Documentation\\File2.html",
    "Software reference");
    // Repeat previous lines for each reference.
    }
END_RUNNABLE_IMPLEMENTATION(ClassName, ClassParentName)

// Proxy

void ClassName::OnOverloadDefaultProxy()
    {
    ClassParentName::OnOverloadDefaultProxy();
    // Here, you can overload the child components
    // OverloadProxy(<NewComponentID>, <Path>, 0);
    // Path is the list of IDs
    }

//Create and destroy

void ClassName::OnCreate()
    {
    ClassParentName::OnCreate();
    //Here, load the needed data
    }

void ClassName::OnDestroy()
```

```
    {
    // Release the data, which were allocated by the previous method.
    ClassParentName::OnDestroy();
    }


void ClassName::OnRegisterParam()
    {
    ClassParentName::OnRegisterParam();

// Here, register your component parameters.
        Add(P1.Init(201301, "Name of parameter", 0.45, GeV, "GeV"));
// Repeat previous line for each parameter.

    }

void ClassName::OnOverloadDefaultParam()
    {
    ClassParentName::OnOverloadDefaultParam();

   //Here, you can overload  default parameter values
   // for child components.
    // OverloadDoubleParam(<NewValue>, <ParameterID>, <Path>, 0);
    // Path is the list of IDs.
    }

// Output

void ClassName::OnRegisterOutputConfig()
    {
    ClassParentName::OnRegisterOutputConfig();

    AddEvent(Event1, 2013001, "Name of event");
    // NMParticle::Config(Event1);
    // NMParticlePtrVect::Config(Event1);
    // Event1.AddDouble(1300, "Centre mass system energy", "Ecms", "GeV");

    }

void ClassName::OnOverloadDefaultOutputConfig()
    {
    ClassParentName::OnOverloadDefaultOutputConfig();
    }

// Input
```

```
void ClassName::OnRegisterInputMaps()
    {
    ClassParentName::OnRegisterInputMaps();

//Here, register input maps and their contents
AddMap(Map1.Init(2013100, "Map1 name"));
    Map1.Add(IP11.Init(2013101, "P11 Name", 2212, "Encoding"));
    Map1.Add(IP12.Init(2013102, "P12 Name", 2212, "Encoding"));
    Map1.Add(IP13.Init(2013103, "P13 Name", 200, GeV, "GeV"));

    AddMap(Map2.Init(2013200, "Map2 name"));
    Map1.Add(IP21.Init(2013201, "P21 Name", 2212, "Encoding"));
    Map1.Add(IP22.Init(2013202, "P22 Name", 2212, "Encoding"));
    Map1.Add(IP13);  // This variable has already been initialized

    aInputMap.Default = 2013100; // default input map
    }


bool ClassName::OnRun(unsigned ParentCounter, NM_ID MapID)
    {
    if (MapID == Map1.ID)
        {
        // Your code
        return true;
        }
    if (MapID == Map2.ID)
        {
        // Your code
        return true;
        }
    return ClassParentName::OnRun(ParentEventCounter, MapID);
    }


// Your code
```

17.3. How to Substitute a Model Component? Let us assume, that we have created the *NiMaX.exe* execution file (see also the main function example). Let us assume, that we have main component with its identifier value *2013* and two child alternative model components, which are registered with the values of their identifiers: *2009* and *2015*, respectively. Let us replace the *2009* component by the *2015* component. At first one should run the command: *NiMax -I 2013* to obtain the file *2013.nmp*, where the *2013* component tree with parameters can be found. Then one should run the command: *NiMax -I 2015* and obtain the file *2015.nmp* with the component parameter tree for the *2015* component. Then within an editor one has to replace the parameter branch *2009* in the *2013* component parameter tree by the *2015*

tree. After that one should run command: *NiMax.exe -R 2013 -i2013.nmi -p2013.nmp* to get generated physical events.

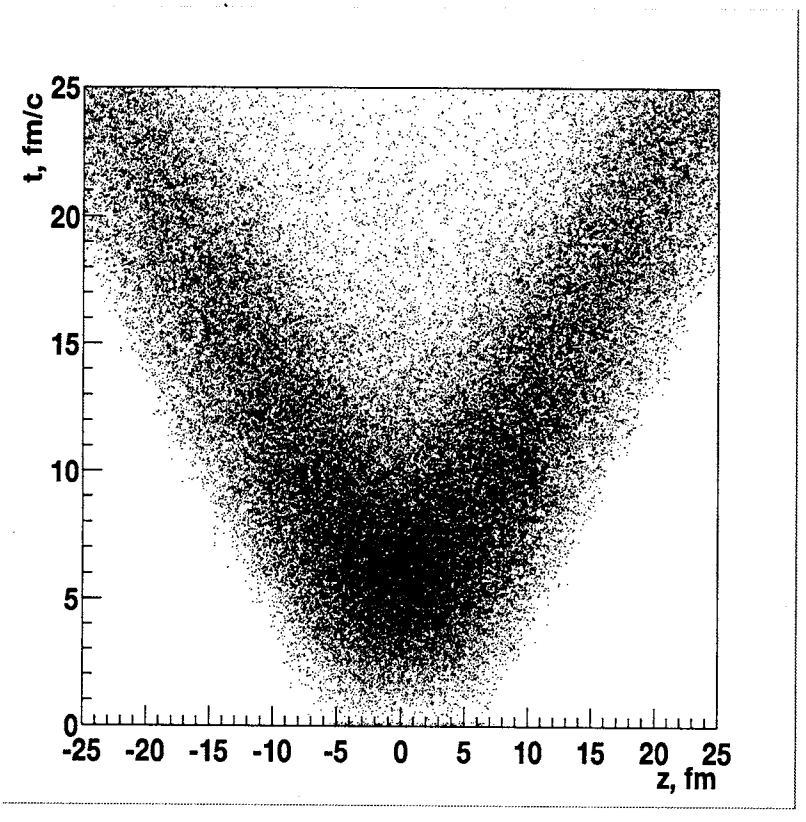

Figure. Space-time distribution of the produced hadrons

17.4. Central Uranium-Uranium Collision Simulation. For the last example we have performed the simulation of central uranium on uranium collisions at the centre of mass energy 6500 GeV per nucleon within the Pomeron based Parton String Model (see physics description in [4]). This model is a complicated model and it was composed of 12 different components.

We have simulated 10 events and created 2-dimensional distribution of the produced particles in space-time (the Z-direction is the beam-direction), which is shown in the Figure. The particle coordinates are calculated at particle freeze-out times. We did not separate the negligible impact of possible spectator nucleons.

References

1. Wenaus T. et al. — GEANT4: An Object-Oriented Toolkit for Simulation in HEP, CERN/LHCC/97-40.

2. Taligent's Guide: Building Object-Oriented Frameworks, *http : //www.ibm.com/java/education/oobuilding/index.html.*

3. Gamma E., Helm R., Johnson R., Vlissides J. — Design Patterns. Elements of Reusable Object-Oriented Software, Eddison-Wesley, 1995.

4. Amelin N. — Physics and Algorithms of the Hadronic Monte-Carlo Event Generators. Notes for a developer., CERN/IT/99/6.

5. Brun R. et al. — ROOT System, CERN/HP, 1997. See *http : //root.cern.ch/.*

6. Amelin N., Komogorov M. — NiMax Manual. Component-Oriented Framework for Hadronic MC Generators, in preparation.

7. Class Library for High Energy Physics, *http : //wwwc1.cern.ch/asd/lhc ++/clhep/index.html*

8. Orfali R., Harkey D., Edwards J. — The Essential Distributed Objects. Survival Guide., John Wiley and Sons, Inc., 1996.

Received on December 29, 1999.