УДК 539.12.162.8

NiMax: A NEW APPROACH TO DEVELOP HADRONIC EVENT GENERATORS IN HEP

N.Amelin, M.Komogorov

The NiMax framework is a new approach to develop, assemble and use hadronic event generators in HEP. There are several important concepts of the NiMax architecture: the component, the data file, the application domain module, the control system and the project. Here we describe these concepts stressing their functionality.

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

NiMax: Новый подход к созданию генераторов событий взаимодействия частиц и ядер

Н.Амелин, М.Комогоров

Обсуждается программное обеспечение (NiMax фреймворк) для разработки, сборки и применения адронных моделей — генераторов взаимодействия частиц и ядер в физике высоких энергий. В данном подходе требуемая сложная модель может быть собрана из отдельных независимых компонент посредством пользовательского графического интерфейса. Особое внимание уделяется обсуждению наиболее важных составляющих данного подхода: компоненты, файла данных, проекта, программной среды, связанной с адронными моделями, и управляющей системы.

Работа выполнена в Лаборатории высоких энергий ОИЯИ.

1. INTRODUCTION

The NiMax framework is a software tool to support a component approach for the hadronic event generator development. This is also a tool to facilitate the work of the event generator users. In our publication [1] we have argued the necessity and advantages to develop such a tool. In that publication we have outlined the framework basic ideas. In [1] we have described the first framework version, which was built as the prototype version, and gave several examples of its application. Recently we have revised this version and made a step forward to real framework adding important mechanism of the component interaction by means of the data file bus. We refer this mechanism as the component collaboration. The model component collaboration allows us to join several model components into a model project. So far we are able to join components into the pipeline projects, where the component execution flow is the same as the pipeline data flow. Below we would like to outline the main features of the NiMax component, the NiMax data file, the NiMax application domain module as well as the NiMax control system. Describing the NiMax framework we would like to pay attention that most of the framework features are not connected with the specific properties of the hadronic event generators domain and our framework has much wider range of its applicability.

2. NiMax COMPONENT

2.1. Component Interfaces. We can consider a component as a set of interfaces. By means of an interface a client can talk with a component asking a definite service. An interface includes several methods or operations and some related data. Let us explain functionality of the standardized NiMax component interfaces, which are presented in Figs. 1, 2, 3.

By means of the *input interface* a user sends a request for a component and provides necessary input data to fulfill this request. A request as well as input data is provided in the form of the input map [1]. The input map is based on the lists of simple data types and has linear structure.

The *tuning interface* gives a possibility of tuning a component with the aim to obtain reliable result from its execution. For example, by means of this interface a user handles hadronic model parameters. Our framework has a set of classes to support parameter and input map management [1].

The result of component execution is obtained by means of the *output interface*. This interface writes the component output data on the data file.

The output data are stored as configured data events and have tree structure [1]. There is a special kind of the data events. We refer them as the predefined events (see below).

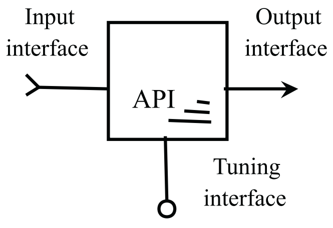

Fig. 1. Main component interfaces

If component starts to run from the request obtained by the input interface we refer it as the main component. Figure 1 shows its example. A component can also read its input from the data file as it is shown in Fig. 3. It starts to run from the request obtained by the *matching interface*.

The matching interface is needed to offer the component collaboration. According to a matching configuration it selects output data obtained from a component to be used as starting input data for another component. The idea of input data selection is illustrated by Figs. 4–5.

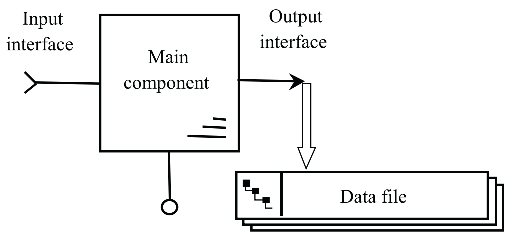

Fig. 2. Component output interface

The written data configuration has linear or tree structure (it will be explained below).

The component matching configurations (matching maps) are realized similarly as the component input maps [1].

A component can have several matching maps. Any of them can be registered as the default one. A component user is allowed to edit them (under the framework control), to

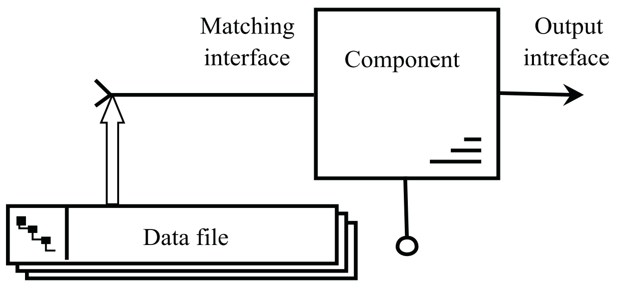

Fig. 3. Component matching interface

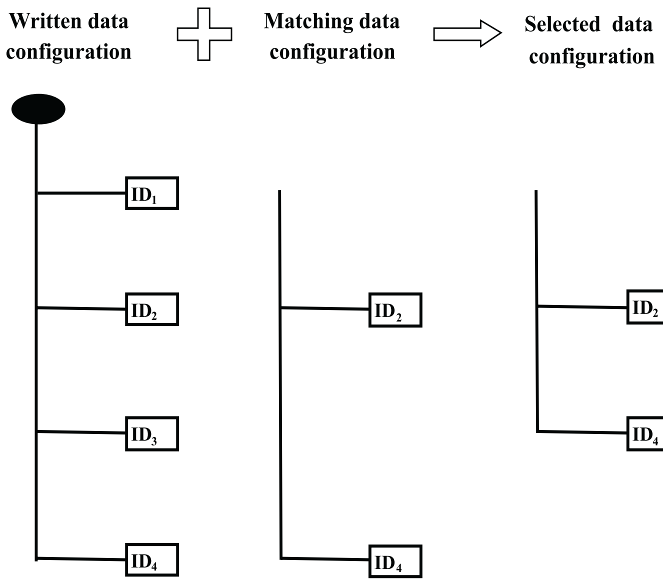
Written data configuration
Matching data configuration
Selected data configuration
ID1
ID2
ID3
ID4
ID2
ID4
ID2
ID4

Fig. 4. Matching the linear structured data

change the default map and to add its own matching maps. The matching configuration represents the configuration of the basic data types. The matching configurations can have either linear data structure or tree data structure.

Each component has its *application programming interface* (API), i.e., a set of public and protected methods, which implement the component functionality and can be called directly as in the case of component aggregation (see below) or indirectly by means of the component interface methods.

2.2. Component Interface Views. Each component interface has at least one view document as it is shown in Fig. 6.

Thus, a component user has the possibility of visualizing component input maps, component matching maps, component parameters and its output configuration.

2.3. Common Component Elements. Besides standardized interfaces and views the framework components have other common elements. Each component has its own component

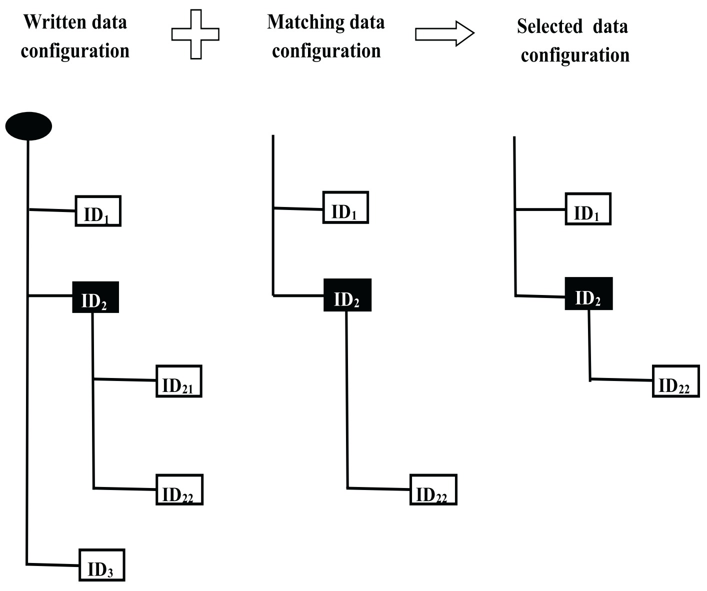
Written data configuration
Matching data configuration
Selected data configuration
ID1
ID2
ID21
ID22
ID3
ID1
ID2
ID22
ID1
ID2
ID22

Fig. 5. Matching the tree structured data

factory to create the component objects and includes the component static information, which is needed for component object creation [1]. Any component has its unique identifier [1]. The knowledge of component's identifier helps us to obtain full information about the component. Particularly, the composite component aggregating (see below) other components should include their proxies, which have the component identifiers as proxy's members.

2.4. Component Inheritance. Each component supports the inheritance mechanism. It is illustrated by Fig. 7.

Thus, we distinguish the base and the derived components. The interfaces of base component and derived component are joined as well as their public APIs.

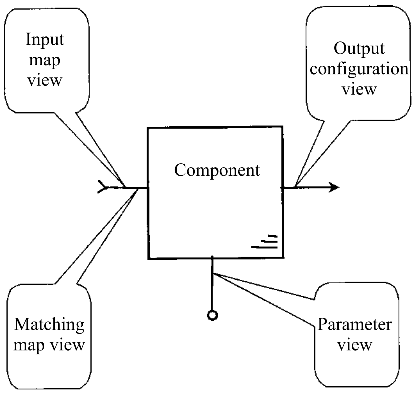

Fig. 6. Component interface views

2.5. Component Aggregation. As shown in Fig. 8 a component can include several aggregated components. These components can belong to different libraries. We refer the aggregating component as the parent component and the aggregated components are considered as the child components. A child component can aggregate other child components. Thus, it can be considered as the parent component relatively to its child components. A component can include the tree of aggregated components.

The child's component tuning and output interfaces are simply joined to the respective aggregating component interfaces.

2.6. Sub-Component Substitution. The inheritance mechanism offers a possibility for component runtime substitution (see Fig. 9). A component can be substituted by another component, if they have common base component (see also [1]). Thus, inside the composite component an aggregated component can be substituted by an alternative component without coding, i.e., using the user interface.

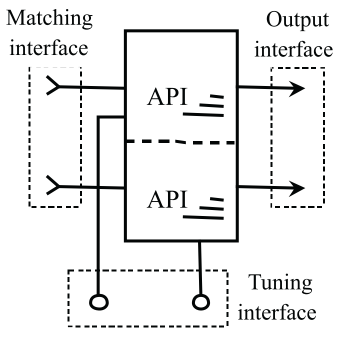

Fig. 7. Component inheritance

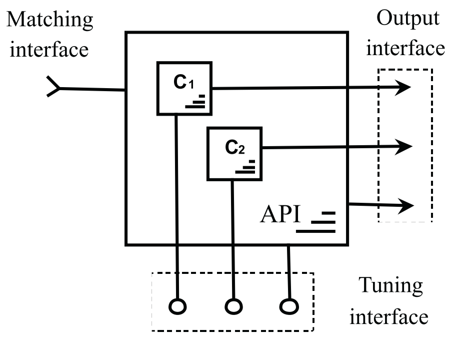

Fig. 8. Component aggregation

2.7. Component Development. To help a component developer we created the component wizard by customizing the Microsoft Visual C++ 6.0 application wizard [3]. The component wizard is a code generator that produces a component skeleton with class names, source code file names, etc., which are specified through the dialog windows. For developers, if they are not going to use the Windows platform, we have created several component frames [1]. These frames help to produce the component skeletons by a text editor.

As was explained above, a new component can be derived from the existing one by means of the inheritance mechanism, e.g., to extend the component applicability. A new component can be developed by aggregation of the existing components. For the last case the child component public API's methods can be called by references.

We would stress that for the developer, which is working on the creation of a component, by aggregation of other components, practically there are no limitations to create an efficient component code, e.g., as compared with the standard C++ coding. In this case the component coding is even simplified. For example, no efforts are required to create and destroy the child component objects. To provide more flexibility for the C++ code developer we have introduced the so-called general and virtual components.

The first one is a component without the input and matching interfaces. Such component is assumed to be only as an aggregated component. The object of a virtual component cannot be created. The interaction between aggregating components is connected with their data exchange. We have developed the data transfer classes [1] (see below) to support the data exchange between hadronic model components.

2.8. Component Collaboration. Besides of the component aggregation, our component can participate in another type of the component interaction: the component collaboration.

The component collaboration is the interaction between components by means of the data bus. For this type of interaction the components are able to collaborate in the situations, when they are isolated from each other. It means either there are no common exchange objects as

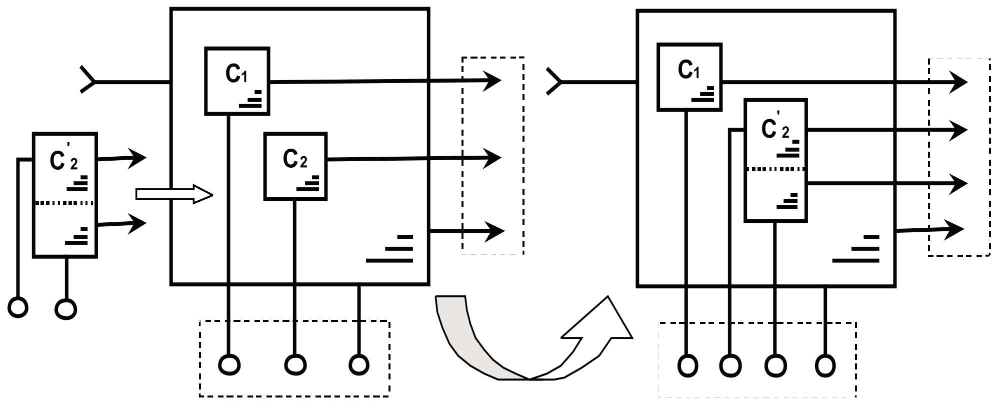

Fig. 9. Sub-component substitution

above discussed objects of the data transfer classes or the components have no common code, e.g., no common INCLUDE files or the components are performed in separate processes, etc. To provide such component independence we have developed the data file as the data bus and the output and matching component interfaces, which were discussed above.

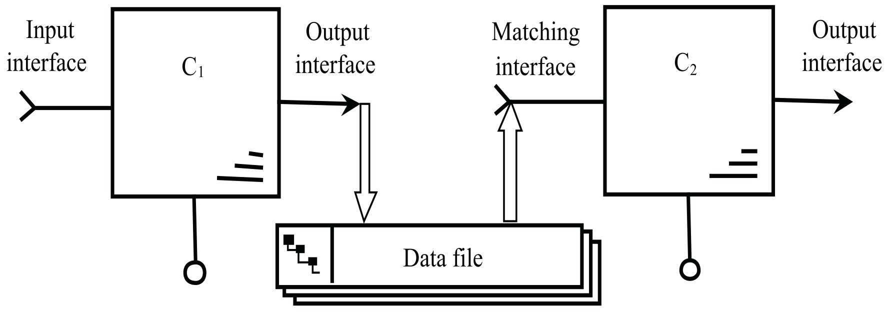

Fig. 10. Component collaboration by the data file bus

3. NiMax DATA FILE

Discussing the component we have already shown an important role of the data file. We can even consider our component as an entity, which can write data by means of the output interface on the NiMax data file and read data by means of matching interface from it.

An essential feature of our data file is that it can be useful also outside the framework. The idea is to write data together with their configuration, which is based on the list of basic data types [1]. Thus, these data can be read within any other package.

3.1. Data File Structure. The data file structure is shortly explained in Fig. 11. Of course, the data file has its header (it is not shown in Fig. 11), which is needed to identify the file and facilitate the navigation through it.

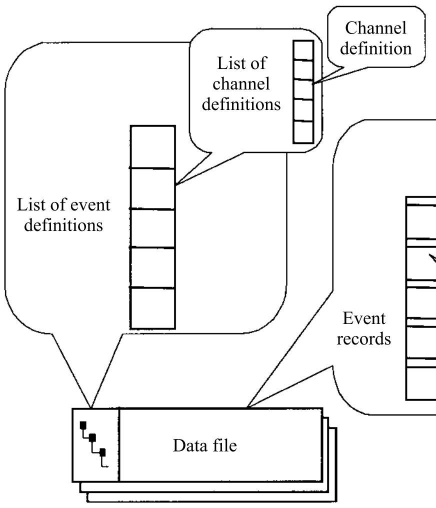

Fig. 11. Data file structure

The data file is separated into two parts: the event configuration part and the event data part. The event configuration part has also its own header and the list of event definitions with the event unique identifiers.

The event configuration header keeps the statistical information about events, e.g., the number of events of the given type. Each event definition includes tree structured channel definitions defining channel identifiers, their types and having more information [1].

As we already explained (see also [1]) a user has the possibility of customizing the event configuration that can be written by a particular component through its output interface view.

The event data part consists of the event records, where the tree structured data are written. Each of them has its own header, where the information to restore the history of the generated physical event [1] and the information to navigate through the data can be found.

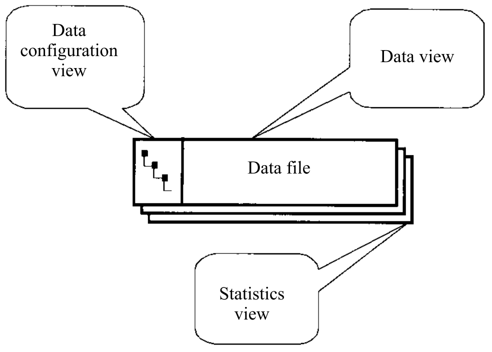

Fig. 12. Data file views

3.2. Data File Views. A possibility of visualizing data file content by means of the data file views is shown in Fig. 12. A user can visualize either written data configuration or written data themselves as well as to visualize some statistical information about the data.

3.3. Predefined Event Views. We suggest to write to the data file the so-called predefined events [1]. These events are prefabricated for definite views within our framework. To deal with such events, the framework control system needs to know only their identifiers (see Fig. 13)).

The component parameter set, the component input maps and the component matching maps as well as the one-, two- and three-dimensional histograms can be stored as such events.

The component predefined events are component states and they can be used to re-execute components.

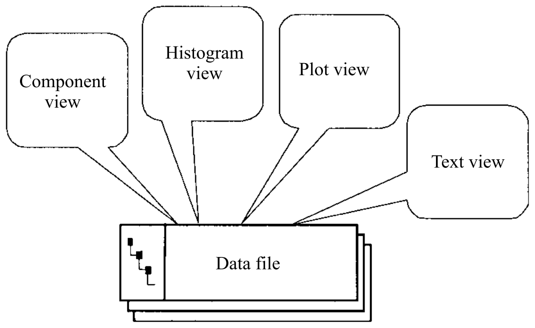

Fig. 13. Predefined event views

4. NiMax APPLICATION DOMAIN MODULES

We should note that the built framework components are not completely independent in the sense that they can use some common data, functions and classes related to the hadronic models. These are physical units, physical constants and physical tables, kinematics functions, data transfer classes [1], etc.

Of course, a hadronic model developer is able to build the independent components, e.g., to develop a component with the built in code physical tables. But it leads to inefficient use of the computer memory and makes difficulties to handle such tables.

Developing the hadronic model related software, which does not belong to the particular components, we had in mind to facilitate the work of a hadronic model component developer (see [1]). Such development is argued by more efficient use of computer resources as well.

Thus, components are packaged into modules in a variety of ways together with their application domain environments.

4.1. Hadronic Components. A large number of the hadronic model components have been already implemented and included into the hadronic model modules [2] (see also [3]) according to their applications.

These modules are prepared as the dynamic link libraries.

4.2. Physical Units and Constants. In this category we include, e.g., definitions of the MeV, the GeV, the barn, the Plank constant, etc. We have adopted convenient strategy to use physical units and physical constants from the GEANT4 [4].

4.3. Hadronic Tables. These tables store the information about physical properties of particles and nuclei as well as hadronic interaction cross sections. Such information is requested in the read-only mode. This fact opens a possibility of handling tables by external tools, e.g., by the external databases.

4.4. Utility Functions and Classes. There are many developed functions, e.g., the kinematics functions, the random number generators as well as many classes, e.g., the integrators that belong to this category.

4.5. Data Transfer Classes. The objects of the data transfer classes are mediated in the data exchange between components (see [1] for more details) in the case of component aggregation. For example, these classes are useful for component data output, for the output of the generated physical event history and to implement universal numerical algorithm [1].

Using the data transfer name we would stress that set of classes providing such functionality can be developed in different application domains. Within our framework we have no strict recommendations how to build such classes. The most important thing is that any object of the data transfer class should support serialization in the sense that it needs methods to write to the data file and read from the data file its object states. Taking into account this fact we have created the data transfer class wizard (similarly as for the component wizard mentioned above) with the aim to help a model component developer.

4.6. Predefined Event Classes. We have developed a set of classes to support the predefined events discussed above. These are the histogram related classes. These classes are similar as the data transfer classes in the sense that they support serialization.

5. NiMax CONTROL SYSTEM

The framework control system has two sides as it is illustrated in Fig. 14. From one side it controls all framework internal processes. On the other hand, it provides a shell between the user interface and the framework parts. Thus, the framework control system methods can be respectively separated into the internal processes related group and the user related group. The framework control system can also be divided into the data file control system and the component control system. Such division makes sense because the data file, the data file views and the data file control system can be joined into the data file system having its own applications. The component control system includes the methods to manage the data file. From this point of view we can consider the component control system as the system, which is derived from the data file system.

5.1. Component Life Cycle. In spite of the fact that the component life cycle consists mostly of internal framework processes, which are hidden from framework user we would like to give an idea about it (see also [1]). There are many possibilities for a user to affect the component life cycle. It includes several phases: the component instance creation phase, the edition phase, the execution phase and the destruction phase.

The control system creates the context of an environment for a component object, which will be created, before to start the object creation procedure. The context defines creation mode, option variables, which are set to default values, the output and input files, if they will be used. A user is able to modify these option variables by means of the user interface, e.g., a user can either set own default parameters, own input and matching maps or suppress some predefined event output or suppress the runtime information output, etc.

There are two special modes of any component (excepting the virtual one) instance creation. The first special mode is the instance creation for only information purpose. This means that a user cannot make any changes of a component object. This mode provides possibility for a user to learn the component structure by the user interface. The second special mode is the debug mode. It gives possibility of creating general component instance as the main one without the permission to execute it. This mode is added in order to debug the component interfaces.

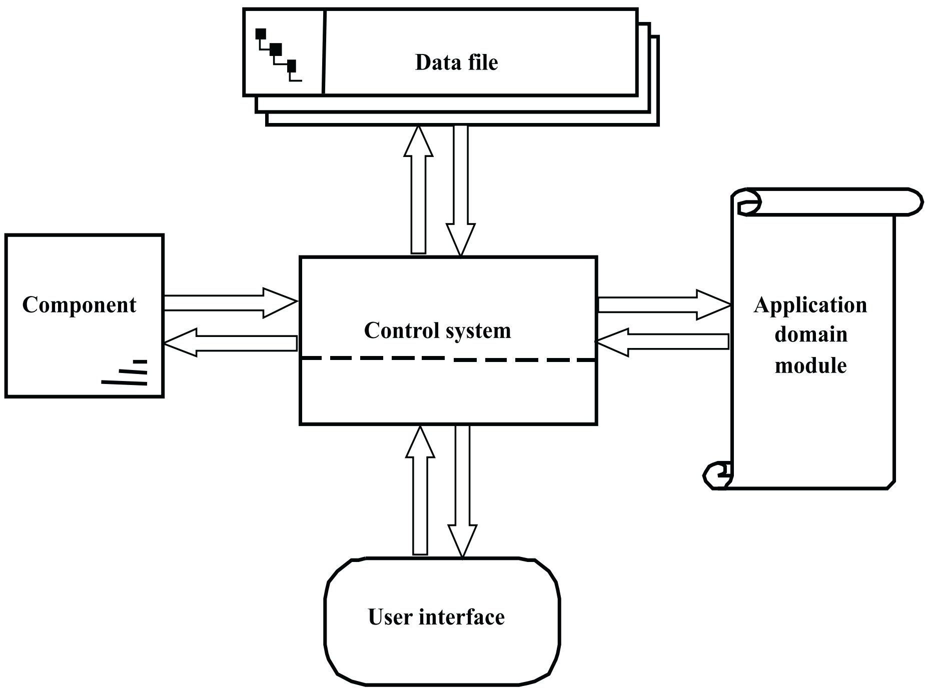

Fig. 14. Control system work

In the case of composite component, the aggregating component instance is created at first. Then control system will create its child component instances. The order of the child instance creation follows their definition order in the parent component. In order to create any component object the framework control system needs to know only the component's identifier. It uses the child component's identifiers to look for their factories. If a factory is not found, the control system tries to find an alternative component according to the component proxy definitions and the component hierarchy. A user is able to control this process changing the context object and enabling or disabling the component substitution. The creation process is repeated for each child component and for their children until all component objects will be created.

The destruction phase for the created component objects is fulfilled in the back order as compared with the construction phase without the user influence.

During the component edition phase a user will be able to edit parameters and input or matching maps as well as reconfigure component's output. The check methods are called to control the consistency of the edition. In the case of nonconsistency these methods send warning messages and set back default values of the non-consistent variables.

During the execution phase the control system supports the component runtime information output: the information messages, warning messages and error messages (see [1]). In the case of an error the control system detects itself the place of the error and a component developer does not need to make special efforts to solve this task.

5.2. Component and Data File Navigation. The tree structure is heavily used in our framework, e.g., the tree structure of the composite components and the tree structure of the data events. Thus the methods to navigate through a composite component and through the data file are similar due to their structural similarity.

Here we would like to mention that using file navigation methods a model developer is also able to write the adapter or driver tools to transform the data written down our data file into the input data, which are acceptable for the external packages.

5.3. Component Librarian. The control system fulfills the component librarian functions. It allows a user to visualize the total list of the components, included into the framework, and to register the required component. Together with the component views it offers a component user the possibility to visualize and handle component input maps, component matching maps, component parameters and component output configurations by the user interface (see also [1]). It checks the edition consistence for the parameters, the input maps and the matching maps. It offers and controls the possibility of substituting a component by an alternative component. For the component, which is assumed to start its run obtaining the needed data from the data file by means of a matching interface, a user has the possibility to navigate through the data file and to perform an attach command to attach the output data file for such component.

5.4. Help Information. Besides the runtime information and the different information messages, which can appear during the component life cycle, the control system offers more detailed help information for the framework user. This information is presented as the html files and can be visualized by any web-browser. The main idea of such help realization is to bind a unique identifier (each component has such identifier, each error message has such identifier, each predefined event has such identifier, etc.) with a html help file [see also [1]). We have also constructed the html file templates to facilitate the work of the component developers.

6. NiMax PROJECT

The component collaboration opens the possibility to join several components into the projects offering an easy way to develop very sophisticated hadronic models from the ready components.

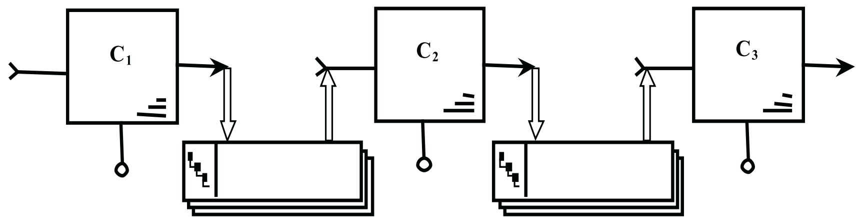

Fig. 15. Pipeline component collaboration

For the particular project a user has to define the component execution flow. The component matching maps should also be registered. So far only the projects, where their components have the pipeline collaboration (see Fig. 15), are allowed.

We would remark that the data files can be loaded in memory improving the project performance and the history of generated physical events [1] will not be lost for the given project, because we keep knowledge from which data event and from which data file the particular component was started.

The simple pipeline project logic limits the set of hadronic models that can be developed. The next step we are working on is to create a project in the situations, when either several components can start their runs from the same data file, as it is shown in Fig. 16 or one component can take its input from different data files.

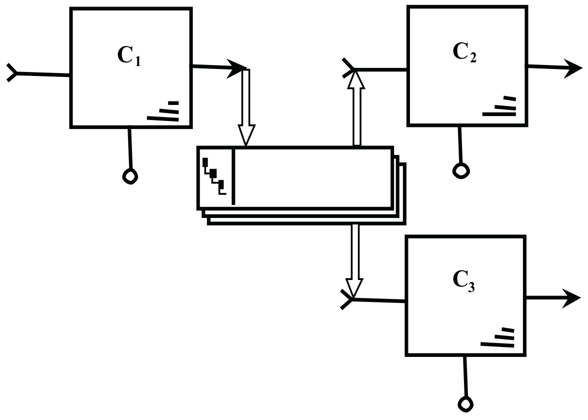

Fig. 16. Tree-type component collaboration

These projects are much more complicated as compared with the pipeline projects due to different reasons. For example, in this case the framework control system should govern the project execution process. Each component, as a result of its execution, produces the data events. The framework control system should dispatch the suitable (according to the component matching maps) events as inputs for other components in a project. The framework control system should offer the project navigation as well. This situation requires also to develop project views in order to select, create, edit and execute projects by means of the user interface.

7. CONCLUSION

We have discussed several important aspects of the NiMax framework. It is a new component approach to develop, assemble and use event generators in HEP. Further work on the NiMax framework is in progress. In conclusion we would like to thank our colleagues and members of the ALICE group at the Jyväskylä University for stimulating discussions. We would express our special thanks for W.Trzaska for collaboration.

References

1. Amelin N., Komogorov M. — JINR Rapid Communications, 1999, No.5-6[97]-99, p.52.

2. Amelin N., Komogorov M. — NiMax Hadronics: Model Components, in preparation.
3. Amelin N. — Physics and Algorithms of the Hadronic Monte-Carlo Event Generators. Notes for a developer. CERN/IT/99/6.
4. Kruglinski D. J., Shepherd G., Wingo S. — Programming Microsoft Visual C++, Fifth Edition, Microsoft Press, 1998.
5. Wenaus T. et al. — GEANT4: An Object-Oriented Toolkit for Simulation in HEP, CERN/LHCC/97-40.

Received on May 22, 2000.