Data Management

Job Configuration
Bookkeeping
Data production

LHCB Technical Note

Issue: Draft
Revision: 1

Reference: LHCB COMP 02-nn
Created: 22nd Feb 2002
Last modified: 16 May 2002

Prepared By: J.Closier, M.Frank, E.van Herwijnen, F.Jacot-Loverre, P.Mato, S.Ponce, A.Valassi
Editor: M.Frank

Abstract

The Gaudi framework, which is used by the LHCb experiment to perform all data processing tasks is highly customisable and though offers large functionality. Once customized, a Gaudi task must be submitted to data processing farms and the resulting files made available to the physicists for further analysis. In this document we discuss the design of components to facilitate this functionality. Special attention was given to allow interaction customisation for different environments as the various data processing farms offer them.

Document Status Sheet

Table 1 Document Status Sheet

1. Document Title: [Project Name Qualification] User Requirements Document			
2. Document Reference Number: [Document Reference Number]			
3. Issue	4. Revision	5. Date	6. Reason for change
Draft	1	17 Feb 2002	First version

Table of Contents

List of Figures

1 Introduction

1.1 Purpose of the document

This document is the result of the design phase for the software and the tools necessary to

- Configure data processing applications for the LHCb experiment
- Access event data to perform the data processing task
- Perform data production tasks such as Monte-Carlo simulation.

The architecture of the software includes the logical and physical structure, which has been forged by all the strategic decisions applied during development.

The other purpose of this document is that it serves as the main material for the scheduled architecture review that will take place in the next weeks. The architecture review will allow us to identify what are the weaknesses or strengths of the proposed architecture as well as to obtain a list of suggested changes to improve it before the system is being realized in code. It is in our interest to identify the possible problems at the design phase of the software project before much of the software is implemented. Essential decisions must be crosschecked carefully with the experts (the reviewers) because they are essential for the system, which is being developed.

1.2 Definitions, acronyms and abbreviations

1.2.1 Definitions

Architecture

The software architecture of a program or computing system is the structure or structures of the system, which comprises software components, the externally visible properties of those components, and the relationships among them.

Framework

A framework represents a collection of classes that provide a set of services for a particular domain; a framework exports a number of individual classes and mechanisms that clients can use or adapt. A framework realizes the architecture.

Component

A software component is a re-usable piece of software that has a well-specified public interface and it implements a limited functionality. Software components achieve reuse by following standard conventions.

1.3 Acronyms

CERN	European Organization for Nuclear Research
UML	Unified Modeling Language
WWW	World Wide Web

1.4 References

[1] User Requirements document

[2] Gaudi Framework

[3] Gaudi Architecture document

1.5 Overview of the document

The first chapter of the document is the introduction containing the purpose, scope and series of definition of terms. Chapter 2 includes the overview of the system together with the description of its basic functionality. The overall system design is described in chapter 3. This system design includes the first level decomposition of the system into hardware and functional components with diagrams. Starting from chapter 4, each chapter is devoted to the description of a single component. It includes what the component is, what is the purpose, what it does, how it is decomposed, the interfaces and the dependencies with other components. Chapter 8 lists the deliverables of the first stage of the project.

2 System Overview

In this chapter we introduce the problem of configuring data processing tasks for the LHCb experiment, which then will be submitted to a data production facility. The output files of this production activity must then be made available to physicists for further data analysis. For more details about individual use cases please refer to the user requirements document [1].

Data processing applications for the LHCb experiment are implemented using the Gaudi Framework [2]. The Gaudi framework consists of various collaboration components, which are centred on so called *Algorithms* extracting the physics content from the analysed events originating from particle collisions. We have categorized the people who work with the tools, our *customers*, into 4 groups. This categorization is not intended to be exclusive; it is a categorization of interaction rather than of people and many people will belong to several groups

Physicists: The “physicist user” develops data analysis code in order to extract physics parameters from the data. To do so he has to instrument his data analysis program and then process files he obtained from the bookkeeping facility. To actually perform the data analysis, the physicist also needs to configure his analysis task using self-developed or stock components. The selected components then have to be configured according to some standard configuration, where he can partially override options. Any software configuration or data output is for private use only. Sophisticated tools may at a later stage take these private configurations into account.

Package Managers: Package managers maintain individual packages, which are part of one or several applications. A package does not only consist of a given version of the software, but also of a set of accompanying parameters, which determine the functionality of the implemented components. Package managers must maintain both. The parameters may even have several versions – depending on their usage in different applications.

Application Managers: Application managers are librarians who deploy a new version of applications – such as the reconstruction program – or parts of it e.g. the tracking package(s). The deployment of a package includes setting up the default configuration options to execute the components of the package. The application manager only picks these settings and releases the application. In order to test the new version of the released code configuration a test must be performed, which requires the execution of a few events.

Data Production Managers: The data production manager supervises and control major data processing efforts on behalf of the collaboration. The data production managers select applications, which were prepared by application managers and submit the necessary jobs to a data processing facility. The data production manager has to monitor the production, and once finished, make the produced files and the relevant production parameters available to physicists for further analysis.

Data ManagementJob Configuration
Bookkeeping
Data production
LHCB Technical Note
Issue: Draft
System Overview
Reference: LHCB COMP 02-nn
Revision: 1
Last modified: 16 May 2002

3 System Design

The architecture of the system is described in terms of the components we have identified and their interactions. A software component is a part of the system, which performs a single function and has a well-defined interface. Components interact with other components through their interfaces.

The notation we are using in this document to specify the architecture is the Unified Modelling Language (UML). We are going to use mainly *object diagrams* to describe a snapshot of the components and their relationships at a given moment in time, *class diagrams* to show the software structure and *sequence diagrams* to describe some of the use cases or scenarios.

3.1 Definitions

This section is devoted to the definition of some concepts and terms that are used in this document. Data processing for a high-energy physics experiment typically is performed in several *steps*: as an example Figure 1 shows a Monte-Carlo event generation *Production.* The events are first generated (GEN), the detector response is then simulated (SIM), the events are reconstructed (REC) and finally further processed (Mini). The concrete steps and their sequencing may vary depending on the data production. Handling of different execution chains is thus necessary depending on optimization criteria. The individual steps may also be executed in parallel in different *jobs*. The job is the minimal entity the system deals with. It typically is a process executing in a processing farm.

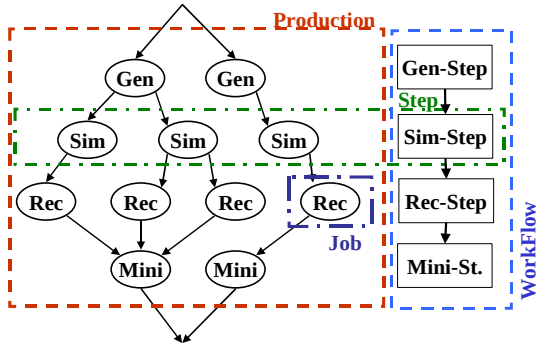

Figure 1 An example of an event processing activity. A *Production* is build out of several *Steps*. A step may be executed in parallel by several *Jobs*.

Production: A production is understood here as the context of a data processing activity, as seen by the user that started it. As an example, it could be the production from scratch of a 25K events mini DST, which includes generation, simulation, reconstruction and mini DST building (see Figure 1). The number of subtasks doesn't matter here. However, there are a fixed number of logical processing steps to be executed. The processing steps to be executed are defined by a workflow. A production could actually be seen as the instantiation of a list of steps to be performed in a given order *within the context of a processing request.*

Workflow: A workflow is defined as a sequence of Steps, which must be executed in a well-defined order to produce a given type of output file from a given type of input file(s).

Step: A Step represents one logical processing unit within a workflow. Several workflows may reuse the same step. A processing step can only accept input files of one given type (e.g. RAHW), but may however produce a variable number of output files, which depends on the configuration. In the previous example, steps are generation, simulation, reconstruction and mini-DST production.

Job: The job is the smallest unit, which the Bookkeeping database deals with. It consists of a task, seen as the database as unbreakable, which has inputs and output files. Steps are typically made of jobs, running on different machines in different places. Jobs themselves could execute in parallel at a lower level, but this will never appear in the Bookkeeping database.

File: Every job consumes and produces data, which typically are files, unless the job is part of the initial data processing chain. Data resulting from particle collisions, event data, which are logically grouped together, represent one file.

3.2 Major design criteria

Before we start with the description of the architecture we have crafted we need to explicitly document what have been our design criteria and what are our strategic decisions.

Seamless integration with Gaudi applications

The LHCb experiment uses the Gaudi application framework for its data processing activities. Any component or tool to be developed should easily integrate into this framework. Adopting the Gaudi design criteria with well defined interfaces of the components will facilitate the integration of the software into Gaudi applications as well as the development of tools, which can be implemented using a light-weight Gaudi application. The clients through interfaces initiate any interaction with the system such as data retrieval or manipulation.

Separation between domains

Whenever very loosely connected domains are identified, it is advisable to separate them. Loose connections can be implemented at the level of tools, which interact with both domains. As an example: the job configuration domain is highly independent from data production and bookkeeping. However, when the data production manager submits a production job, the configuration must be extracted from the job configuration. This interaction should be implemented at the level of the production tools.

Implementation of Tools

All functionalities required by the users will be implemented in tools, which in turn use the different domain services through the interfaces. This approach allows us to develop independently new tools, more sophisticated tools or simple specialized tools without affecting existing functionality, because all tools interact with the domain services through the exposed abstract interface. The domain services may even implement new functionality without affecting existing functionality as long as the existing interfaces do not change.

Integration technology standards

The second fundamental design principle of Gaudi implies that technology choices are hidden. Only the service knows about a specific technology, the client stays technology independent. This principle will be adopted. As an example, all domains require persistent back-end solutions. At the same time some of the components to be developed may have to be instantiated in several production centres or even individually by single users wanting a private, non-proprietary persistent back-end. We are neither in the position to impose a persistency choice nor do we want to lock ourselves into one technology. For this reason, we try to follow the ideas behind popular integration technologies such as ODBC (C/C++ binding), JDBC (Java binding) or wxODBC (Python), so that later adoption of new products will not be traumatic. A very similar argument of course is valid for the choice of a suitable GUI toolkit.

3.3 Domain Decomposition

After analysis of typical use-cases, which were collected from representative members of the different envisaged users, four different domains were identified:

The Configuration domain

This includes both the configuration of the software in terms of packages, libraries and executables, and the parameters used at run-time, which define the behaviour of the software. Whereas package and application managers actively edit these data, physicist users and production managers access this database in read-only mode to extract or view “default” application configurations.

The Bookkeeping domain

The production manager populates this database whenever newly created files should be made publicly available. All other users only request information about certain files, which could be looked up. This includes also the search for files available for physics analysis as well as the possibility to browse more detailed information concerning the history and the production parameters/conditions of the file in question.

The Data Production domain

Only very few users will interact with the data production environment: this area typically is under the control of data production managers, who perform data reprocessing and Monte-Carlo data generation for the benefit of the entire collaboration. It is understood, that persistent information is kept only temporary until a given production is finished. Then the production managers, with the help of a tool, will publish the files and migrate all relevant information to the bookkeeping area, where it will be accessible to physicists and package/application managers. The production database can then be safely flushed.

Job Submission domain

Only the data production manager will interact with this environment. Concrete implementations may exist in several flavours – depending on the facilities and the environment available for data processing. Concrete implementations will e.g. deal with lsf, nqs, use PVSS for monitoring etc. This component will also deal with GRID specific implementations. Clearly the implementations are least specified and must be highly adaptable to the environment.

Figure 2 shows the interaction of the different users with the domains. Although there is an apparent dataflow between the domains, there is no direct data flow across domains unless through the usage of tools. The domains and their functionality will be encapsulated by this mechanism.

The access of the domain data through the use of interfaces and tools also provides an independent and incremental upgrade path for sophisticated tools. Whereas first implementations may be rather crude, such as command line tools, any level of sophistication using graphical user interfaces may be achieved without any change required for the underlying access mechanism, which is solely performed in the service.

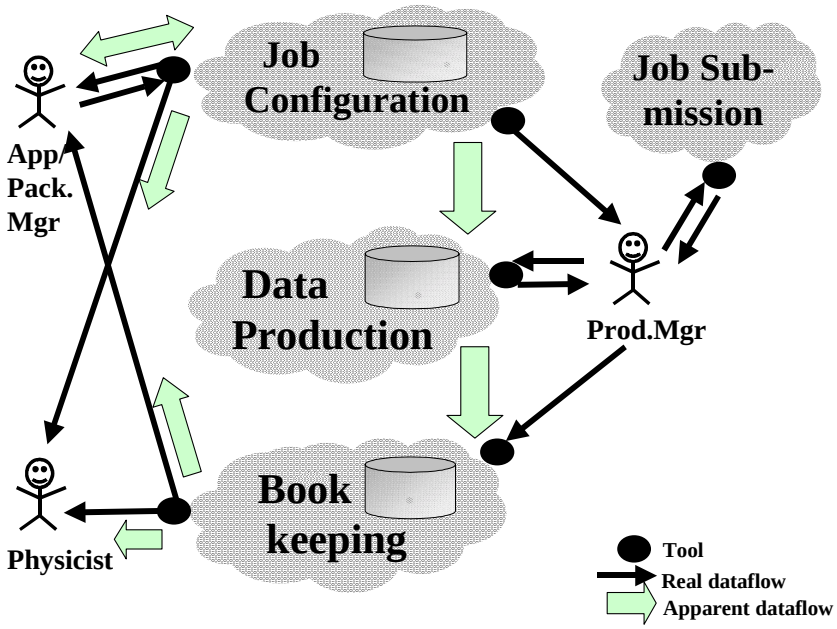

Figure 2 The users interacting with the different domains. The interaction is through tools, which offer the requested functionality.

3.4 Object Diagram

We introduce the description of the architecture by an *object diagram* showing the main components of the system (Figure 3). We know that object diagrams are not the best way to show the structure of the software but they are very illustrative for explaining how the system is decomposed. They represent a hypothetical snapshot of the state of the system, showing the objects (in our case component instances) and their relationships in terms of navigability and usage.

Application Manager

As in any standard Gaudi application the *application* manager sits at the top of the hierarchy. The *application manager* creates all services needed by the tools. It is essentially this functionality, which allows him to integrate the resulting services into the Gaudi framework. The tools are nothing but a special lightweight instantiation of a Gaudi application.

3.4.1 Services

Services implement the basic functionalities required by the different domains. The services should be kept as simple as possible and as complicated as necessary. Typically the services implement mostly atomic actions and data manipulations. Complicated functions are decomposed into the implemented atomic functionality and implemented by the tools (see section 3.4.2).

Job Configuration Service

The job configuration service is capable of programmatically editing and managing Gaudi job options and other relevant parameters to execute a Gaudi task.

Bookkeeping Service

The bookkeeping service is capable of programmatically access public event data files. This service is also capable of accessing hooks necessary to retrieve the production information from the job configuration service. A separate interface allows it to add/edit entries in the bookkeeping database, which correspond to data files.

Data Production Service

The data production service is capable of programmatically access to the information stored in the data production database. This service permits as well to define and manipulate workflow.

Job Submission Service

This service is capable of programmatically submitting, controlling and monitoring jobs on a data processing facility. This service will strongly collaborate with the Data Production Service, which provides all information necessary for job submission.

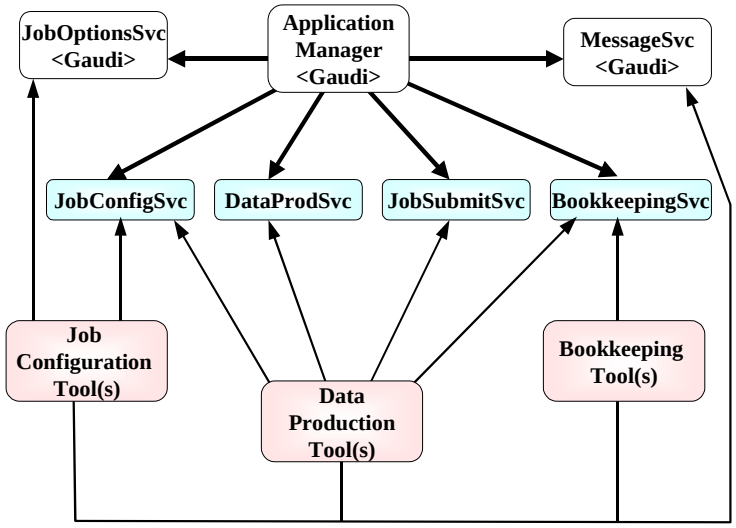

Figure 3 The object diagram shows the services implementing the basic functionality and the tools interacting with these services. Services may reuse existing services from the standard Gaudi framework.

3.4.2 Tools

Tools are an essential concept of the design. Whereas the services implement atomic actions, the tools must support all functionalities required by the users of the system. Tools may interact with only one or several domains and can implement any level of sophistication. The concrete development of tools depends on the customer e.g. physicists possibly want to interact in a different way with the job configuration than application managers.

For a first implementation we will not envisage full-blown tools with graphical user interfaces etc. Simple command line oriented tools should be sufficient to prove that the design was functionally complete. Later tools specialized for the different customers will be developed.

Tools typically have three flavours:

- Tools to browse information from the various domains using the web. Such functionality typically has to be implemented at the level of the web server or the supporting database.
- Tools to extract information from a given domain. These tools typically are command line oriented. An example is the extraction of the options to configure the reconstruction program of a given version.
- Tools to edit domain data. This may be done interactively using GUI based tools, from the command shell or from the scripting prompt. This functionality already suggests that some tools are implemented as Gaudi services themselves.

Job configuration tool(s)

The tools to edit the job configuration database do this solely through the interfaces offered by the job configuration service. Application and package managers edit the parameters. Physicists and Production managers will take tagged configurations of these parameters.

Bookkeeping tool(s)

Bookkeeping tools permit to access and edit the bookkeeping information. Only data production managers use tools to edit the bookkeeping information. All other customers only have read access to these data.

Data production tool(s)

The data production tools only have one customer: the data production manager. The tools simplify his work to configure, submit, monitor and control data productions such as Monte-Carlo simulation or reprocessing. These tools are likely to interact with all domains, as they must be accessed (see Figure 2).

Possibly at a later stage some of the tools used by the data production manager can be made publicly available to facilitate e.g. remote job submission to GRID enabled facilities.

3.5 Components and Component Model

Having introduced the decomposition of the system in the previous section, it is important now to show the structure of the software in terms of classes we are envisaging and their hierarchies. At this level we also want to introduce the different interfaces these classes offer to clients. Any service, typically implemented by a class as shown in Table 2, may implement one or several interfaces – depending on the requested functionality. Table 2 shows the different interfaces and the envisaged functionality.

Table 2 The different interfaces, service classes, tool classes and their basic functionality

Interfaces	**Functionality**
IJobConfigurationInfo	Interface defining the data access to the job configuration information
IJobConfigurationEditor	Interface defining the edit capabilities of the job configuration information
IBookkeepingInfo	Interface defining the data access to bookkeeping information
IBookkeepingEditor	Interface defining the edit capabilities of the bookkeeping information
IWorkFlowInfo	Interface to access work flow related information
IWorkflowEditor	Edit workflows, which are templates for multiple productions
IProductionInfo	Interface to access information of submitted jobs
IProductionEditor	Edit information of productions to be submitted
IJobInfo	Interface to allow Gaudi components to publish data items
IJobControl	Interface to submit/control jobs to a data processing farm
Service	**Implement the functionality of interfaces**
JobConfigurationSvc	Facilitate access to the job configuration information
BookkeepingSvc	Facilitate access to the bookkeeping information
DataProductionSvc	Facilitate access to data production related information
JobInfoSvc	Service to allow Gaudi components to publish data items
JobControlSvc	Implements job submission, monitoring and control functionality. Several implementation may exist depending on the technology used: DataGRID, etc.
Tool Services	**Implement command line functionality to allow implementation of scripts**
JobConfigUI	Implement functionality to interact with the job configuration
DataProductionUI	Implement functionality to interact with the data production environment
BookkeepingUI	Implement functionality to interact with the bookkeeping domain

Data ManagementJob Configuration
Bookkeeping
Data production
LHCB Technical Note
Issue: Draft
System Design
Reference: LHCB COMP 02-nn
Revision: 1
Last modified: 16 May 2002

4 Job Configuration

The job configuration tasks are implemented in the *JobConfigurationSvc*. This service implements all interfaces to edit and access the job configuration data. Any client, such as tools or any other Gaudi task can access this service if needed.

In this chapter we will describe the functionality of this component, the interfaces it offers to other components and its dependencies.

4.1 Purpose and Functionality

The purpose of the job configuration domain is to supply the Gaudi job options service with all options required by other components of the Gaudi task to execute a selected job. It is assumed that facilities allowing the user to edit the options and to save or retrieve sets of them for future use are supplied as well. The options may consist of:

- Job options that override default properties of the algorithms
- Job options that override default properties of the services
- Setup parameters such as small script fragments, environment variables and other parameters needed by an application to bootstrap

Options within a Gaudi application can uniquely be identified by the tuple (*Client Name*, *Option Name*) and each of these tuples a separate *Option Value* can be assigned. Options are consumed by a Gaudi application at start-up of the job, when all components get created. Gaudi offers several possibilities to process job options:

- ASCII text files with C++ like syntax
- Python scripts
- Direct access from a database (currently not implemented)

In a typical application such as a reconstruction program many options [$O(10^2\text{-}10^4)$] could be set separately. The management of all options, which belong to a given application is non-trivial and strongly influences the run-time behaviour. Package coordinators that already manage code packages will have to manage the options, which correspond to the code components. Since code components are re-usable in several applications, different option sets may be defined corresponding to different behaviours.

A database of pre-configured applications acts as a repository for all members of the LHCb collaboration to access standard (pre-configured) applications. Options for one single application or one package within an application can be extracted, edited and updated.

Besides component properties, applications may depend of certain environment variables. Within Gaudi the use of environment variables is not necessary. External software may however depend on environment variables. The environment of an application will also be managed in the job configuration database.

4.2 Definitions

Options: An Option is one set-up parameter of a Gaudi component.

Tagged option configuration: Set of tagged options as they are needed by one package used by an application.

Tagged code configuration: Set of tagged code configurations as they are needed by one package used by the application. The code configuration is a snapshot of the (CMT-) packages used by the application.

Package: A *Package* consists out of a tagged code configuration and a tagged option configuration. A package contains both the code and the setup data necessary to configure and execute the code fragment. For every tagged code configuration several packages may exist – depending on different setups.

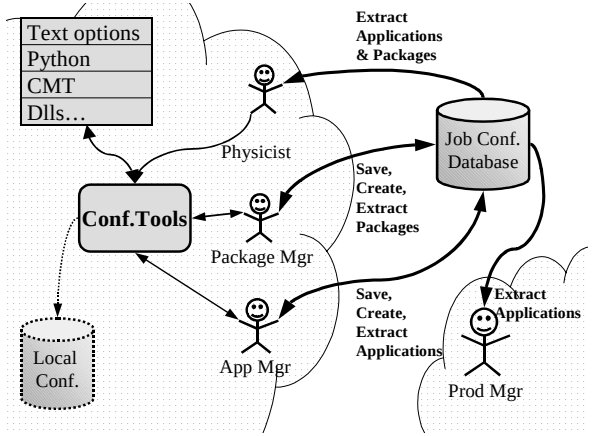

Figure 4 The different user's interaction with the job configuration in order to access or edit data necessary to configure Gaudi tasks.

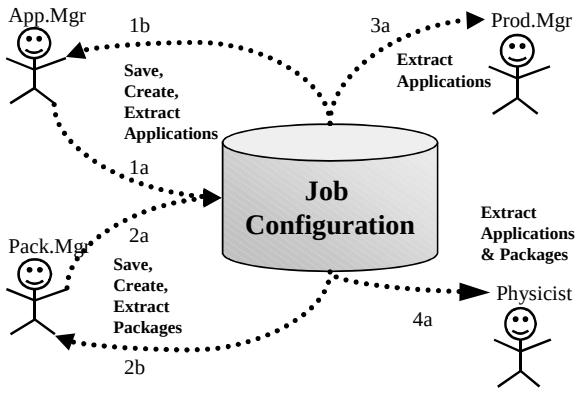

Figure 5 The flow of data between the different customers and the job configuration.

Application: An application is defined as a set of packages. In the same way the application uses the code of the packages it consists of, the options of the application are the aggregation of all options of the packages it consists of. For one code configuration of an application many option configurations may exist – depending on the functionality of the application. An example could be a simulation program, which is able to generate Monte-Carlo events with different detector configurations. In addition an application has a set of parameters attached to it, which could be small scripts or environment variables.

4.3 Interaction with the Service

The following features must be accessible through the job configuration database:

- Extract/save job options and other production parameters for one application
- Extract/save job options and production parameters according to one package used within one application

Note that we can distinguish here two categories of users:

- Application and package managers must provide information.
- Physicists and production managers only retrieve information from the configuration database.

Tools are used to facilitate access to the data stored in the job configuration database. The simplest tools just extract/import the job options of one application or the job options of one package within one application as text file. For full-blown application development the configuration tool may become as complex as needed with embedded graphical user interfaces etc.

Application managers manage the configuration of a complete application

- Save/create the options of one application as a set of packages
- Retrieve the options of one application for modification
- Save/create production parameters for one application
- Retrieve production parameters for one application

Package managers deal with the job options of one package

- Save/Create the options of one package
- Retrieve the options of one package

Production managers deal with job options of one application

- Retrieve the options of one application
- Retrieve production parameters for one application

Physicists

- Retrieve options of one application.
- Retrieve options of one package within an application.
- Retrieve production parameters for one application.

4.4 Transient Data Model

The logical data model as shown in Figure 6 describes the connectivity between the identifiable applications and the packages. Please note that the logical data model only describes the logical connection between the objects. The implemented data model shown in Figure 7 does not implement these associations. To resolve the associations, clients must collaborate with the service.

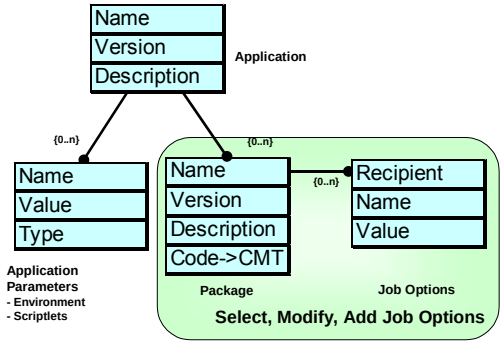

Figure 6 The data model of the job configuration application.

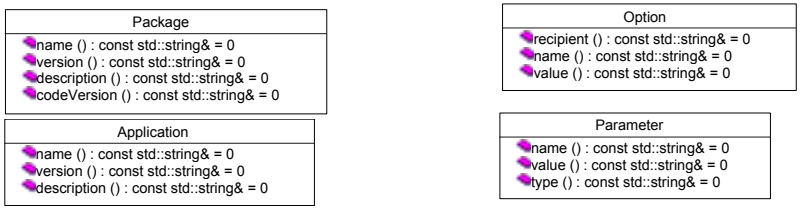

Figure 7 The design of the data classes, which clients retrieve from the service. Note that object associations are not implemented in the class design. Associations are resolved by the service.

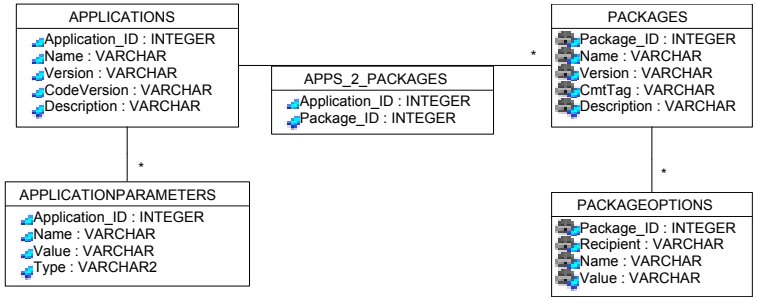

Figure 8 The design of the database tables supporting the service as a persistent back-end.

The described model is based on the following assumptions:

- Application Managers build applications. A new application consists of packages, which in turn have options. Environment and other parameters are assigned to the application directly. These are obtained e.g. by a call to CMT.
- Packages and applications may exist in several versions for one code (CMT) configuration. The code configuration stored in each package entry allows later identifying the tagged code version to be used for this package.
- Packages may be reused for several applications
- Job parameters are *not* reused by several packages

4.5 Persistent Data Model

The persistent data model describes the design of the data not as clients do see them, but rather as the database, which is necessary to store all the configurations sees them. When using a relational database this is its design. Figure 8 shows the design of the database tables, which can support the service. Associations are resolved by matching a SQL query between the corresponding integer object identifiers and the corresponding foreign keys. Due to the reuse of packages by several applications, a separate link table is used to model this association.

4.6 Interface Model

The job configuration is accessible to any Gaudi application using the Standard Gaudi mechanism:

- A Gaudi service encapsulates the access to the database. Accessible interfaces implement all the required functionality to edit applications, packages, component options and application parameters.
- No database internal details are exposed. Objects passed to clients do not expose database internals.
- Links do not have to be filled immediately. The service resolves on request object dependencies such as the associations between application objects and package objects.

The encapsulation of the functionality as a Gaudi service allows to independently enhance and develop any sophisticated user tools.

The user interaction model addresses and reflects all actions identified in the use cases. The service must offer access to the data as well as edit capabilities of the data. These two functionalities are reflected in separated interfaces, both implemented by the service. This split allows access restriction independently of the database's access rights.

Job configuration interface (*IJobConfigurationInfo*): This interface allows performing the following actions as identified in the use-case analysis (All actions only require read-only access to persistent data):

- Retrieve all applications available.
- Retrieve all options for a given application identified by its name and version.
- Retrieve environment parameters, which correspond to one application identified by its name and version.
- Retrieve all packages available.
- Retrieve all packages used by a selected application
- Retrieve a new package.
- Retrieve all options for a package.

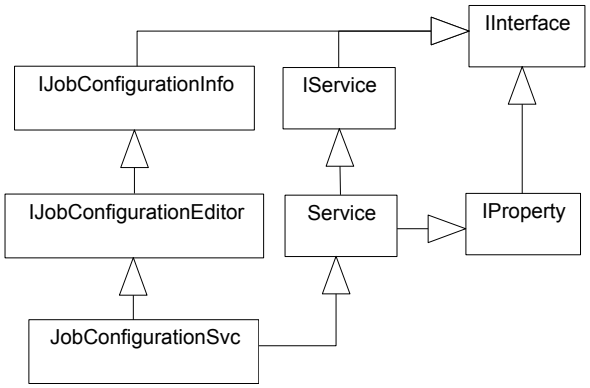

Figure 9 The class diagram of the designed service component with all interfaces.

IJobConfigurationInfo
application (app_nam : const string&, app_vsn : const string&, appH : AppHandle) : StatusCode = 0
applications (apps : std::vector<AppHandle>&) : StatusCode = 0
applicationOpt (appH : const AppHandle&, opt_act : const string&, opt_nam : const string&, optH : OptHandle&) : StatusCode = 0
applicationOpts (appH : const AppHandle&, opts : std::vector<OptHandle>&) : StatusCode = 0
applicationOpts (appH : const AppHandle&, opt_act : const string&, opts : std::vector<OptHandle>&) : StatusCode = 0
applicationParam (appH : const AppHandle&, par_nam : const string&, parH : ParHandle) : StatusCode=0
applicationParams (appH : const AppHandle&, pars : std::vector<ParHandle>&) : StatusCode = 0
applicationParams (appH : const AppHandle&, type : const string&, pars : std::vector<ParHandle>&) : StatusCode = 0
packages (pkgs : std::vector<PkgHandle>) : StatusCode = 0
package (pkg_nam : const string&, pkg_vsn : const string&, pkgs : PkgHandle&) : StatusCode = 0
packages (appH : const AppHandle&, pkgs : std::vector<PkgHandle>) : StatusCode = 0
packageOpt (pkgH : const PkgHandle&, opt_act : const string&, opt_nam : const string&, optH : OptHandle&) : StatusCode = 0
packageOpts (pkgH : const PkgHandle&, opts : std::vector<OptHandle>&) : StatusCode = 0
packageOpts (pkgH : const PkgHandle&, opt_act : const string&, opts : std::vector<OptHandle>&) : StatusCode = 0

IJobConfigurationEditor
createApplication (name : const string&, vsn : const string&, appH : AppHandle) : StatusCode = 0
deleteApplication (name : const string&, vsn : const string&) : StatusCode = 0
modifyApplicationDescription (appH& : AppHandle&, desc : const string&) : StatusCode = 0
createPackage (name : const string&, vsn : const string&, pkgH : PkgHandle&) : StatusCode = 0
deletePackage (name : const string&, vsn : const string&) : StatusCode = 0
modifyPackageCmtTag (pkgH& : PkgHandle&, code_tag : const string&) : StatusCode = 0
modifyPackageDescription (pkgH& : PkgHandle&, desc : const string&) : StatusCode = 0
createAppParam (appH : const AppHandle&, par_name : const string&, par_value : const string&, par_typ : const string&) : StatusCode=0
modifyAppParamValue (appH : const AppHandle&, par_name : const string&, par_value : const string&) : StatusCode=0
deleteAppParam (appH : const AppHandle&, par_name : const string&) : StatusCode = 0
createPackageOpt (pkgH : PkgHandle&, opt_actor : const string&, opt_nam : const string&, opt_val : const string&) : StatusCode = 0
modifyPackageOptValue (pkgH : PkgHandle&, opt_actor : const string&, opt_nam : const string&, opt_val : const string&) : StatusCode = 0
deletePackageOpt (pkgH : PkgHandle&, opt_actor : const string&, opt_nam : const string&) : StatusCode = 0
addPackage (appH : AppHandle&, pkgH : const PkgHandle&) : StatusCode = 0
removePackage (appH : AppHandle&, pkgH : const PkgHandle&) : StatusCode = 0

Figure 10 The detailed description of the interfaces implemented by the job configuration service.

Job configuration editor interface (*IJobConfigurationEditor*): This interface offers the functionality to modify, edit, add and delete data. Hence the usage of this interface requires write access to the persistent data. Features are:

- Retrieve/edit/delete/create an application
- Retrieve/edit/delete/create a package.
- Retrieve/edit/delete/create environment parameters, which correspond to one application identified by its name and version.
- Remove/edit/delete/create each option of a given package

Properties interface (*IProperty*). This standard Gaudi interface allows the application manager to customize the behaviour of the JobConfigurationInfo interface by setting new values to component's properties.

Service interface (*IService*): This standard Gaudi interface is implemented by every Gaudi service. It ensures the proper service is initialisation and finalization by the application manager.

The class design of the component is shown in Figure 9 and Figure 10.

4.7 Dependencies

The job configuration service depends on the following services provided by other components of the architecture:

- The *MessageSvc* is used to print error messages.
- The *ApplicationMgr* must instantiate and initialise the job configuration service.
- The database access is performed using the *GaudiDb* package. Using this package also facility e.g. to extract the job configuration data corresponding to a given application, save the extracted data e.g. in MS Access format and use the extracted information as a local sub-database.

5 Bookkeeping

The bookkeeping tasks are implemented in the *BookkeepingSvc*. This service implements all interfaces to edit and access the bookkeeping data. Any client, such as tools or any other Gaudi task can access this service if needed, as long.

In this chapter we will describe the functionality of this component, the interfaces it offers to other components and its dependencies.

5.1 Purpose and Functionality

The bookkeeping domain is the final managerial unit handling the access to data produced by the various data processing activities. Files are the objects the clients are interested in. They are mapped to files, which are the basic objects the Bookkeeping domain deals with. There is however, a rather large amount of additional information linked to an existing file, which is of interest to the clients and is in a way part of a file:

File information

- Logical file name
- Number of events contained in the data file
- Production date
- File size in Bytes
- File type
- Object content

Job configuration information

- Global options e.g. production channel ID
- Options used to create the file
- Options of previous processing step

Quality information

- Individual analysis groups may “sign” a given file as being useful for their analysis.
- The data manager may flag a given flag as checked/non checked

All this information must be accessible.

Note that on top of this “file centric” or “file centric” system, Gaudi provides a way to access and select on an event base by using specialized NTuples called Event Tag Collections. Figure 11 gives an idea of the architecture of these Tag Collections. More details can be found in the Gaudi User Guide.

5.2 Definitions

Quality: The quality information is a way to “sign” a given file as being useful (or useless) for a given analysis. It may thus consist of a flag telling whether the file is usable and of a recipient describing which type of analysis this flag is applicable to. The content is anyway free thanks to the very open architecture.

File types: The description of a file type is mainly the description of the content of the file. Examples are “log file”, “DST version 3”, “RAW data version 2” or “cards file for a given algorithm”.

5.3 Interaction with the Service

The interactions of the different users with the Bookkeeping service are shown on Figure 12. Here is an overview of it:

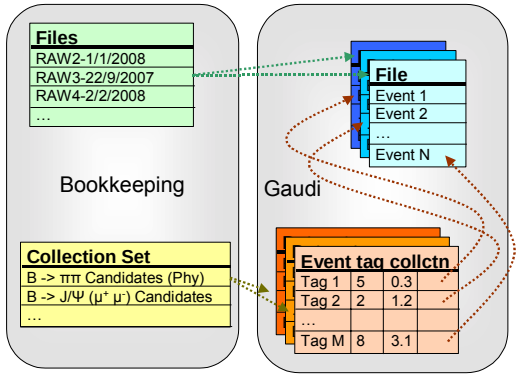

Figure 11 The Gaudi model to access event data.

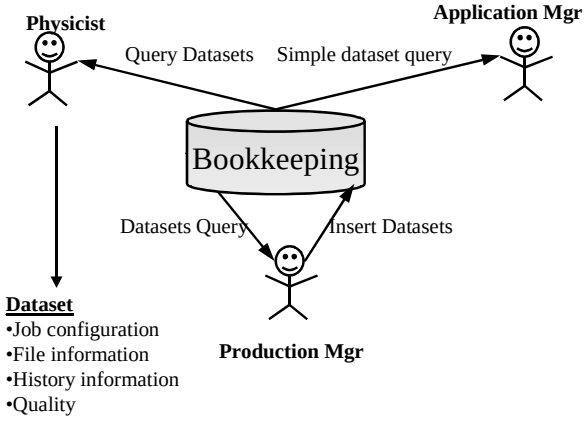

Figure 12 Interaction diagram for the bookkeeping database

Physicist: Physicists only retrieve files or related information. For them a file is not only a logical file, but also includes all related information such as event generation parameters etc. Since this group of users is by far the biggest, special attention must be given to the presentation of the information of a file.

Application Manager: Application managers only issue simple queries for files. They do not care which events they receive for testing applications, but simply want to receive a number of events of a given type.

Production Manager: Production managers are the only persons, who write to the bookkeeping database. They only require read access e.g. in the event of reprocessing.

5.4 Transient Data Model

5.4.1 Data Model

The logical data model as shown on Figure 13 describes the connectivity between the identifiable jobs, the files containing the data and the quality flags. Please note that the logical data model only describes the logical connections between the objects. These associations may not be implemented in the actual data model and one may have to use a service to resolve some associations.

The described data model is based on the following assumptions:

- The bookkeeping system deals mainly with jobs and files and
  - Every file is the output of a unique job but may be the input of many jobs
  - A job may have several files as input and several files as output
- Each job is described by
  - A date corresponding to the submission date
  - A pointer to the corresponding item in the configuration database. This pointer actually provides the configuration name and version of the application run by this job, as given in the configuration database. This provides the description of the job configuration, including default job options
  - A set of job options that are incremental changes to the default set of job options coming from the configuration database
  - A set of parameters, given as name/value/type triplets that characterises the job. Examples of the use of parameters are the production location, the run number (in case of real data) or the data source (experimental or simulation). Examples of parameter types are “environment variable” or “script”.
- Each file is at least characterised by
  - A logical name. This name is not a file name by itself but a way to retrieve the file, using for example the Data Grid tools.
  - A file type, which is actually described as a set of parameters. Some parameters are mandatory: the actual type (DST, RAW, Tag collection, Log …) and the version of this type. Others are not. Note that types are shared among many files and that they are versioned. One implication of this last point is that there is only one RAWH or DST type and several versions of it, currently known as RAWH2, RAWH3, ...
  - A set of quality flags, which are actually described as a set of parameters, as for types. Examples of these parameters are the actual flag (good, bad, test …) and the group concerned. Note that quality flags may be shared among many files
  - A set of parameters that characterises the file. Examples of parameters are the number of events in the file, the file size or a pointer to the master copy of this file.

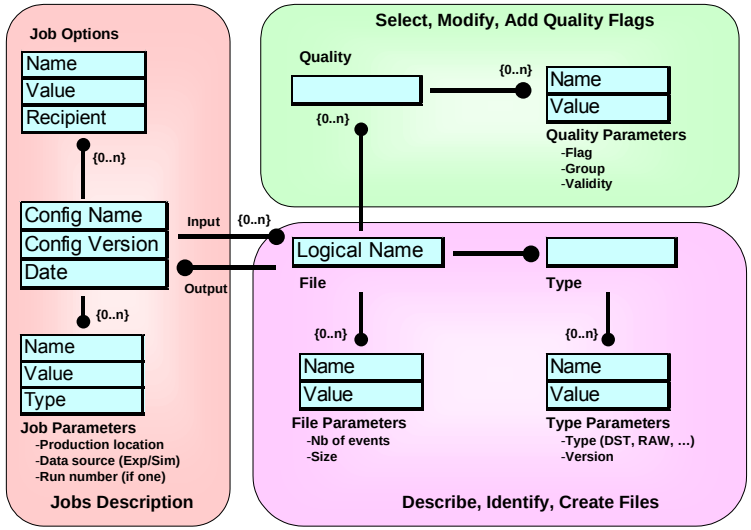

Figure 13 The data model of the Bookkeeping application

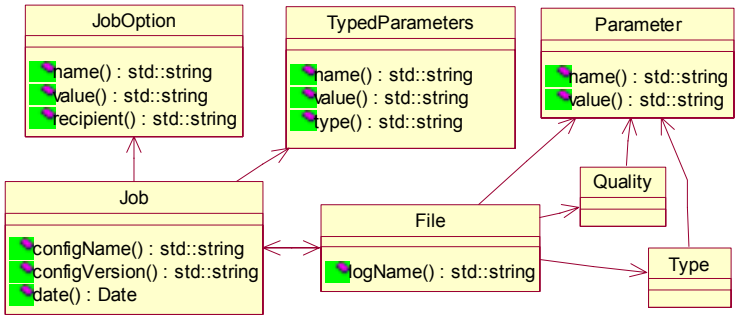

Figure 14 Class diagram from the Bookkeeping application

5.4.2 Class and Object diagrams

The class diagram described on Figure 14 is deduced from the data model of Figure 13. It is slightly different since the Parameter class is reused for every parameter definition. The arrows there simply mean “uses” and do not imply that one class actually points to another.

This class diagram maps to the object diagram shown on Figure 15. This demonstrates the actual representation of the bookkeeping data in memory, with every bubble being an object and every arrow a relation between two objects.

5.5 Persistent Data Model

The persistent (Figure 16) data model describes the design of the data not as client do see them but rather as the database, which is necessary to store the bookkeeping data sees them. In the case of a relational database, this would be the design of the different tables used. The associations drawn as lines show where IDs are supposed to match. As an example, the Job_ID in _Job and in _JobOptions are supposed to match.

The whole design described previously was intended to build thin tables on the database side by removing all columns that would not be absolutely necessary for all cases. These columns are replaced by lines in one of the parameter tables. As an example, the number of event of a given file is only relevant in the case of a file containing data (not in the case of a log file). Thus, is was not added in the _Files table but will be a line in the _FileParams table with name “NbEvent” and value the actual number of event of the associated file.

This architecture also allows taking advantage of the indexing capabilities of the underlying database to search for objects. As an example, indexes should be created in the _InputFile table, based on the file_ID and in the _Files table based on the Job_ID. This allows an efficient browse of the hierarchy of jobs and files.

In principle, every object in the data model maps to a table and every relation between objects to an ID. There are however some exceptions:

- Types are not mapped to a table since this table would have had no column except the ID one. They only exist through IDs in other tables. As an example, the Type_ID is used in the _TypeParams and in the _Files tables.
- Qualities are not mapped to a table. However, the table would have had two columns: one for File_ID and one for Quality_ID. For optimization purposes, it was decided that the File_ID would be replicated in every line of the _QualityParams table and that the _Qualities table would be suppressed. However, all the quality parameters defined for a given file stay grouped by the Quality_ID and will map to parameters of the same Quality object in the transient world.
- The many to many link that exists for input files requires a specific link table: _InputFiles. Each line of this table describes one link between a job and a file, which is seen as an input file of this job. Note that the output files relation is a regular relation and is thus mapped to an ID, namely the Job_ID in _Files.

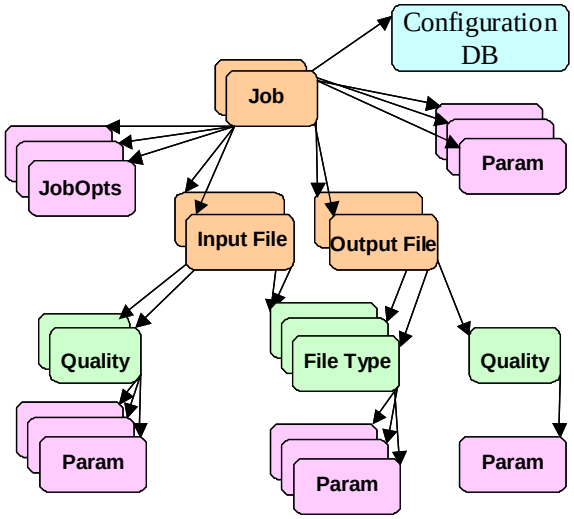

Figure 15 Object diagram for the bookkeeping application

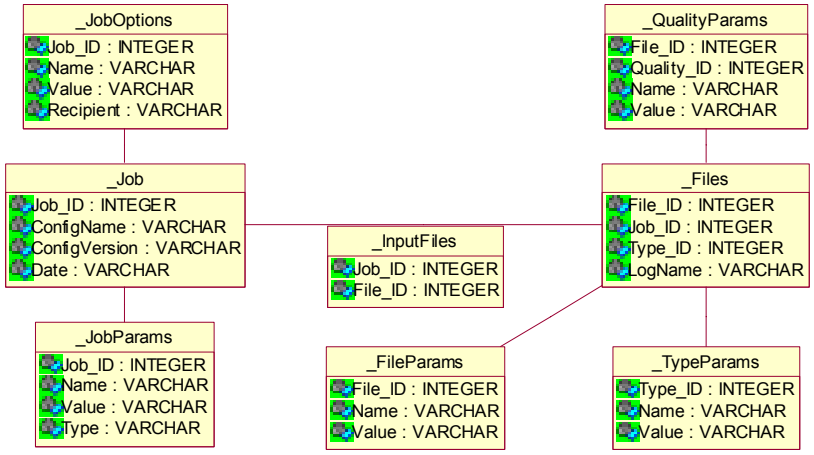

Figure 16 The design of the database tables supporting the service as a persistent back-end.

5.6 Interface Model

The Bookkeeping is accessible to any Gaudi application using the standard Gaudi mechanism:

- A Gaudi service encapsulates the access to the database. Accessible interfaces implement all the required functionality to edit jobs, files, qualities, types and all sorts of parameters.
- No database internal details are exposed.
- Links between objects are not filled at creation time. The service resolves on request object dependencies such as the associations between files and jobs.

The service must offer access to the data as well as edition capabilities. These two functionalities are reflected in two separated interfaces, both implemented by the service. This allows to restrict access to the data independently of the database's access rights.

Bookkeeping interface (IBookkeeping): this interface enables to perform the following actions:

- Retrieve all jobs available, all jobs between two given dates, all jobs with a given configuration, all jobs having a given input file or the job that created a given file
- Retrieve all files available, a given file from its logical name, all the input files of a given job, all the output files of a given job or all files of a given type. One more method allows to retrieve all files fulfilling a given SQL query. This should only be used by experts knowing what they are doing.
- Retrieve all qualities of a given file
- Retrieve all types available or the type of a given file
- For each type of parameter (including jobOptions), retrieve all parameters for a given object or retrieve the parameter with a given name for a given object. For jobOptions parameters, the recipient must be supplied too.

Class design shown in Figure 17

Bookkeeping editor interface (IBookkeepingEditor): This interface allows performing the following actions:

- Create/delete a job, a file, a quality or a type.
- Add/remove input files to/from a job
- For each type of parameters (including jobOptions), create, update and remove a parameter on a given object.

Class design shown in Figure 18

Specialized interfaces could be defined in addition to suit specific needs. We do not know yet which ones.

Properties interface (*IProperty*). This standard Gaudi interface allows the application manager to customize the behaviour of the BookkeepingSvc by setting new values to properties.

Service interface (*IService*): This standard Gaudi interface is implemented by every Gaudi service. It ensures the proper service is initialisation and finalization by the application manager.

The class design of the component is shown in Figure 19.

Figure 17 IBookkeepingInfo abstract interface

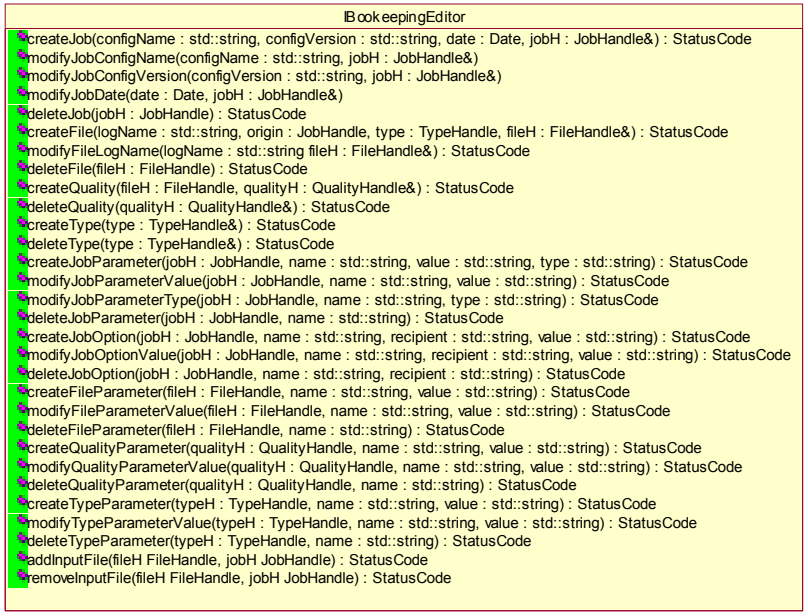

Figure 18 IBookkeepingEditor abstract interface

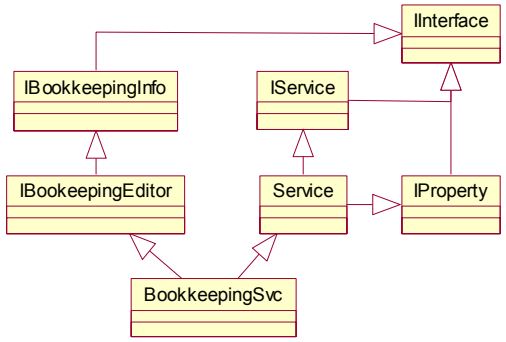

Figure 19 The class diagram of the designed service component with all interfaces.

5.7 Dependencies

The bookkeeping service depends on the following services provided by other components of the architecture:

- The *MessageSvc* is used to print error messages.
- The *ApplicationMgr* must instantiate and initialise the bookkeeping service.
- The database access is performed using the *GaudiDb* package.

6 Data Production

The data production domain supports large-scale data processing activities. Physicists typically trigger such activities, like the production of Monte-Carlo events of a given type or a data reprocessing. The request is then processed by the production manager, who instantiates the necessary jobs and finally publishes the produced files in the bookkeeping database. From then on the physicists can analyse the produced data. The data production domain will support also data taking activities.

In this chapter the interfaces used by this domain and the planned implementation is discussed.

6.1 Purpose and Functionality

The data production domain manages data processing activities, which the production manager executes on behalf of the collaboration. Such productions typically are requested by physicists who give input information such as the type of data to be produced, the configuration of the detector or, in the event of Monte-Carlo production, the event type. The production manager analyses the request and defines the workflow necessary to fulfil it. The workflow will contain all information necessary to produce the requested output data. In the example on Figure 20, a Monte-Carlo data production workflow is described: a generation step is followed by a simulation step, a reconstruction step and finally a processing step to compress the produced data. The production manager will instantiate this workflow, which can be seen as the plan of the work to be done, and create the necessary jobs. Finally once all the data are produced, they will be published.

The above description involves two actions:

- First a workflow must be created or selected. Such a workflow consists of several steps, where each of the steps is characterized by a set of parameters and job options. Each step consumes a set of input data streams and may produce a set of output data streams. On creation of such a workflow, the production manager must be able to
  - Create and edit workflow objects
  - Create and edit the steps, which build up the workflow. A program using a predefined setup to be executed typically characterizes each step: the step must allow the navigation to its configuration information.
  - Assign and remove input and output data streams
  - Create and modify all the options and parameters of workflows, steps and data streams, which characterize these objects.
- Then this workflow must be instantiated and the corresponding jobs submitted. The resulting structure is very similar to the workflow:
  - The workflow object becomes a production object. A run defines a logical group of data processing tasks realizing a workflow. Parameters present in the description of the workflow object are inherited.
  - Each step in the workflow is realized in to one to many job objects. The job is the concrete data processing entity. It is however, not always advantageous to process all events in one job. It may be useful to split up the events into several jobs. How many jobs are actually instantiated and submitted depends on the split criteria shown in Figure 21. Two parameters are used here: Nevt(split) and Nevt(Merge), being the maximum and minimum number of events that could be processed in a single job. The job inherits all parameters and options from the step as well as the input and output stream types. However, it is not sufficient to just connect stream types, but rather concrete data sources, which typically are identified by their logical name in the bookkeeping database.
- As long as the production is not finished, the data production manager wants to monitor the progress. In particular he would like to know:
  - The overall status of the production(s), which reflects the combined state of all jobs executing in the same context.
  - Which jobs are finished, crashed or failed?

The actions described will be implemented in a Gaudi service and the implemented functionality can be accessed through interfaces. The design of this service and its interfaces will be described in the following.

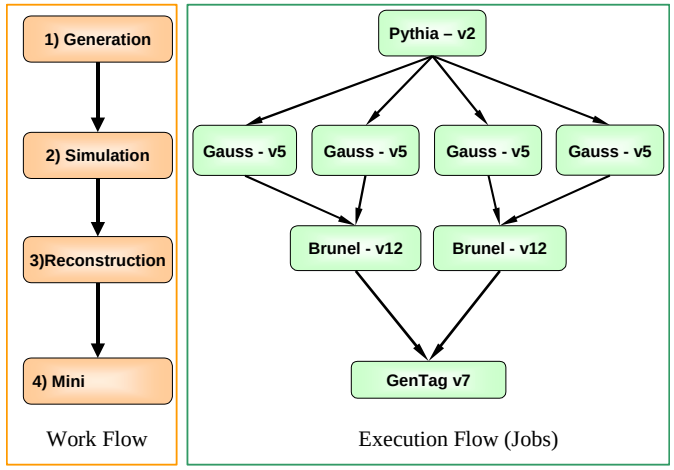

Figure 20 The definition of a workflow and example of an execution flow

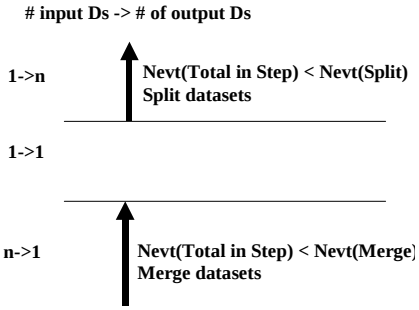

Figure 21 The definition of the criteria to split a step into several parallel executing jobs.

6.2 Interaction with the Service

The object diagram shown in Figure 22 shows all entities, which describe the workflow and the execution flow. According to this diagram, the service must facilitate the production manager to create/modify/edit and delete these objects.

Assuming a physicist made a request for data production, the production manager will interact in the following manner with the production domain:

- Create or use an existing workflow
- Create/modify/edit/delete workflow objects
- Create/modify/edit/delete the steps, which build up the workflow
- Create/modify/edit/delete input and output data streams
- Create/modify/edit/delete all the options and parameters of workflows, steps and data streams, which characterize these objects.

Once the workflow is build, the production manager instantiates the work flow by creating a new execution flow. He will:

- Create/modify/edit/delete production objects
- Create/modify/edit/delete job objects
- Create/modify/edit/delete file objects.

After a data production is created and the jobs execute, monitoring information must be provided. The service must allow access to

- The state of the executing productions.
- The state of the executing jobs.
- In the event of a failure further access to the parameters and options in use by each job.

Whereas the object creation will in most cases require an action by the production manager himself, the update of the objects, such as the state of the production or the jobs does not necessarily involve human action, but may be done by monitoring processes.

6.3 Data Model

The data model as shown in the object diagram in Figure 22 and the class diagram in Figure 23 describes the connectivity between the entities of the workflow and the execution flow. Figure 24 shows the design of the database tables, which can support this service.

The workflow model:

- A workflow is identified by a name. A set of parameters is attached to each workflow object.
- Steps describe each process to be executed in the context of a workflow. Each step is identified by a name. Each step receives its configuration in terms of parameters and job options from the job configuration area. Some parameters such as the key of the job configuration information are mandatory. A step may be re-used by several workflows. However, some parameters like the job split criteria depend for example on the event type and hence are an attribute of the link between the work flow and the step.

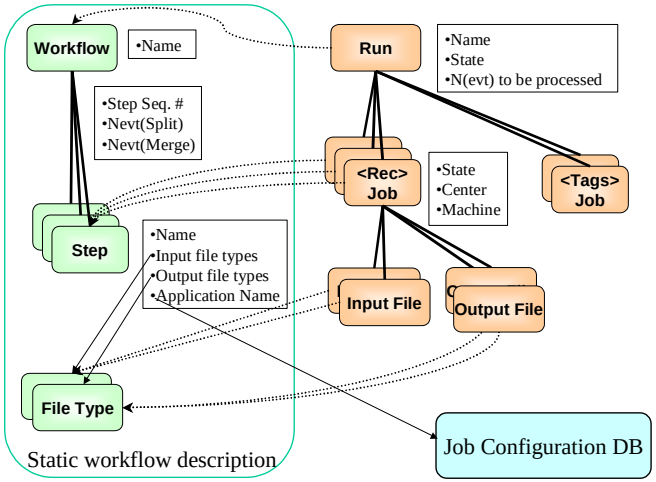

Figure 22 The logical data model representing the different stages. The objects the dashed lines point at act as a source of information to populate the origin objects.

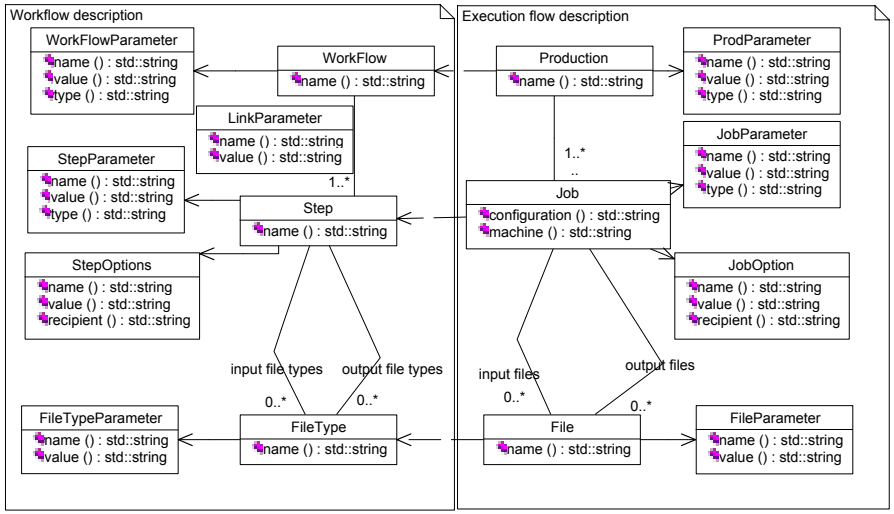

Figure 23The class diagram deduced from the object diagram in Figure 22.

Data ManagementJob Configuration
Bookkeeping
Data production
LHCB Technical Note
Issue: Draft
Data Production
Reference: LHCB COMP 02-nn
Revision: 1
Last modified: 16 May 2002

- Processes described in one step consume and produce files of a given type, which must be known to the production domain. For example a step describing event reconstruction consumes data produced by a simulation step; it would not make sense to process data produced by the event generation step, since required information is missing. File types are versioned, the version being a parameter of the type. Additional parameters may include the object identifiers of the items written to the file for each event (e.g. the identifier of the transient objects in the datastore, "/Event/MC/Particles").

The execution flow:

An execution flow of a data production activity is represented by a production. A production is the instantiation of a workflow together with supplementary information such as the concrete definitions of the files consumed and produced by the executing jobs or the total number of events to be processed. For this reason the parameters and options of objects in the execution flow are not identical to those in the workflow, but typically a superset. Each production is a collection of several jobs. The mapping of a step into jobs is defined in the workflow by the maximum and minimum number of event processed by one job. If there are more events than the maximum, the job is split into several jobs. In case, there are fewer events than the minimum, several jobs may be merged into a single one. Jobs consume and produce files. Productions, Steps and files are described by a number of parameters such as:

- Production
  - The total number of events to be processed
  - The state of the production
- Job
  - A date corresponding to the submission date
  - Job state such as "queued", "running", failed", …
  - The sequence number of the job within the step
  - A pointer to the corresponding item in the configuration database. This provides the description of the job configuration, including default job options
  - A set of job options that are incremental changes to the default set of job options coming from the configuration database
  - A job may have several files as input and several files as output
  - A set of parameters, given as name/value pairs that characterises the job. Examples of the use of parameters are the production location, the run number (in case of real data) or the data source (experimental or simulation)
- File:
  - Any file is the output of one unique job but may be the input of many other jobs
  - A unique logical name. This name requires interpretation before usage, which may include the retrieval of the remote data for example when the job is executing in the context of the Data Grid
  - A file type described as a set of parameters. Mandatory parameters are the file type (such as RAW, DST, MINI) and the version of this file type.
  - A set of parameters that characterises the file. Examples of parameters are the number of events contained in the file, its size or the identifier of the master copy.

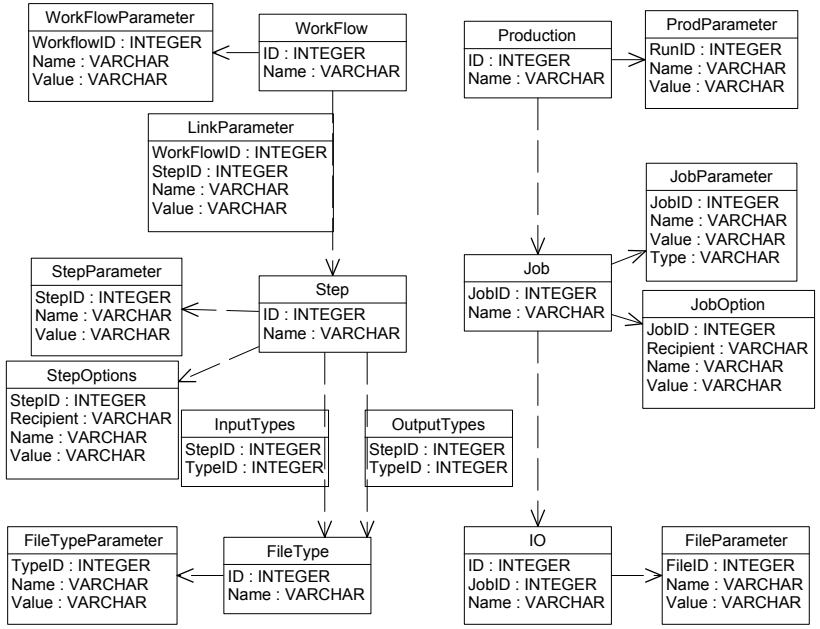

Figure 24 Database tables

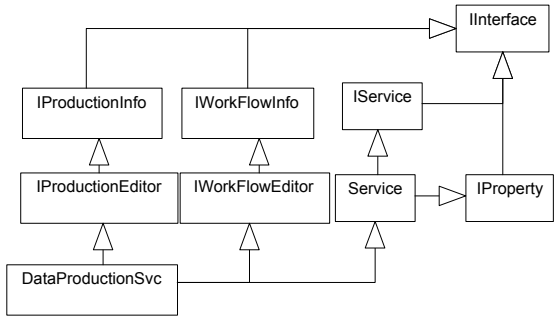

Figure 25 the class diagram of the designed service component with all interfaces

6.4 Interface model

The production information is accessible by any Gaudi application using the standard Gaudi mechanism:

- A Gaudi service encapsulates the access to the database. Accessible interfaces implement all the required functionality to create all sort of production.
- No database internal details are exposed.

The service must facilitate all interactions described in section 6.2. Four rather disconnected activities can be identified and are reflected in the interface design, which is shown in the class diagram in Figure 25:

- Edit workflows: interface IWorkFlowEditor
- Access workflow information: IWorkflowInfo
- Edit productions: IProductionEditor
- Access production information for monitoring or investigations: IProductionInfo

The **Workflow Information Interface** (IworkFlowInfo, see Figure 29) allows to perform the following actions:

- Retrieve all existing Workflow objects
- Retrieve all existing Step objects
- Retrieve all file type objects
- Retrieve for each workflow:
  - The corresponding Step objects and the corresponding link parameters
  - The parameters and options connected to each step
  - Retrieve the input/output file types and their corresponding parameters

The **Workflow Editor Interface** (IworkFlowEditor, see Figure 28) allows performing the following actions:

- Create new workflow, new step and new type.
- Define the steps building the workflow
- Define parameters of a workflow
- Define parameters of a step with respect to the workflow (Link parameters)
- Add/modify/remove job options for a given step
- Add/modify/remove parameters for a given step
- Add/modify/remove input or output type for a given step
- Add/modify/remove a given file type

The **Production Information Interface** (IproductionInfo, see Figure 27) allows to access:

- The state of the executing productions.
- The state of the executing jobs.
- Parameters of each production, job and file object

The **Production Editor Interface** (IproductionEditor, see Figure 26) allows to:

- Create/modify/edit/delete production objects
- Create/modify/edit/delete job objects
- Create/modify/edit/delete file objects.

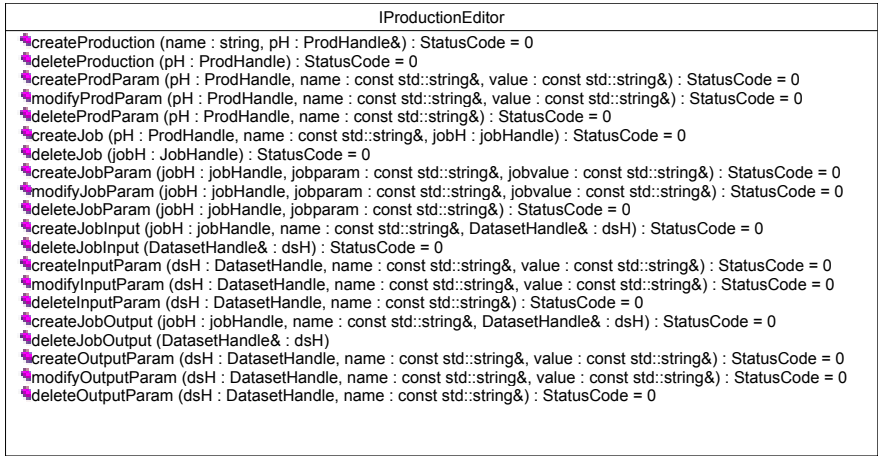

Figure 26 IProductionEditor interface

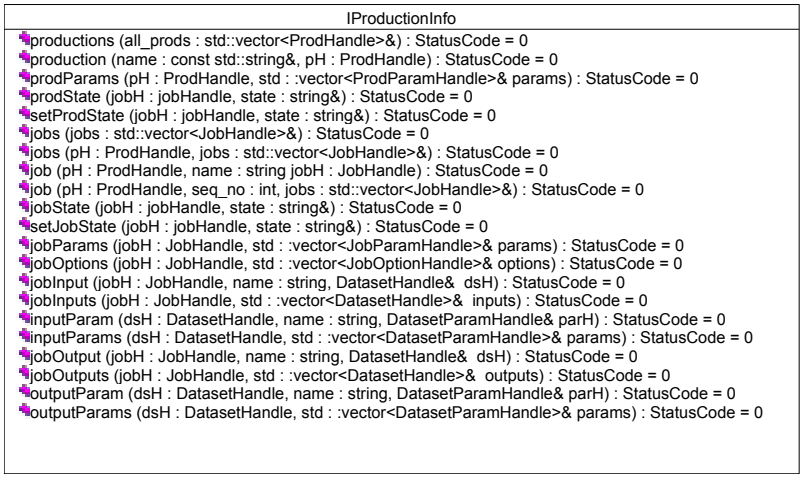

Figure 27 IProductionInfo interface

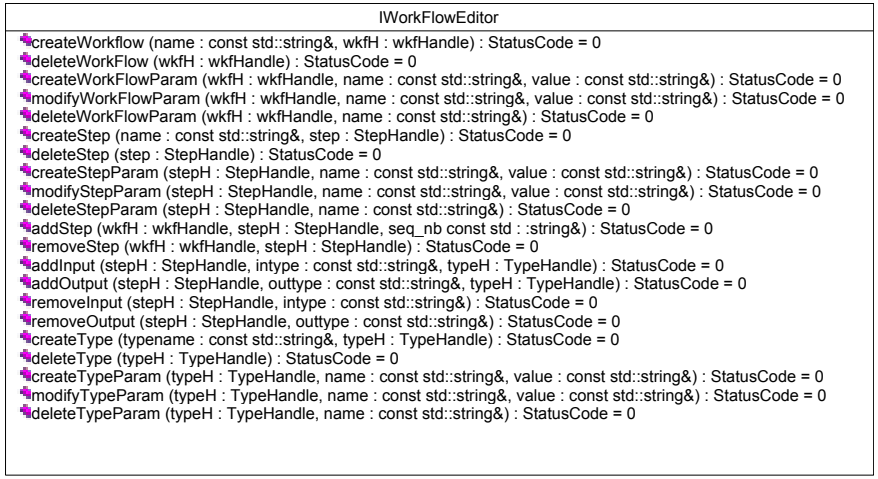

Figure 28 IWorkFlowEditor interface

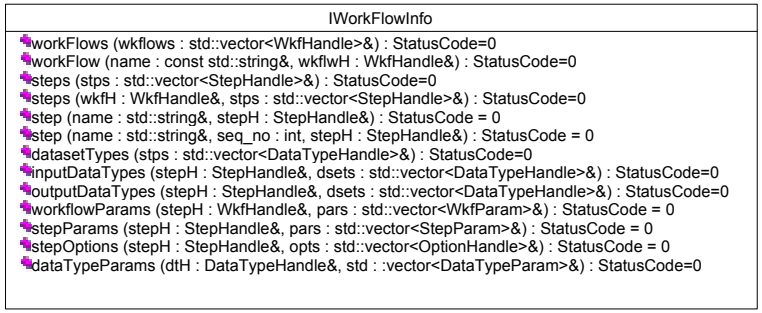

Figure 29 IWorkFlowInfo Interface

7 Job monitoring and control

The job monitoring and control system[1] allow:

- Jobs to send messages to a monitoring system, thus enabling their progress to be tracked via a User Interface
- The monitoring database to update the production database
- Sending messages to jobs, thus allowing the publication of specific information e.g. histograms

[1] Please note, that currently the LHCb DAQ group has started a project to monitor and control tasks for the high level trigger environment. Both systems have very similar functionality and since both systems (hopefully) use similar technologies we hope to benefit from this effort. For this reason, this chapter should only be seen rather informational.

7.1 Purpose and Functionality

With jobs running on a Grid it will become increasingly more important to keep track of running jobs, and to have tools that allow at a glance inspection and problem determination.

The purpose of the job monitoring system is to provide, for every running job, information to the outside world. This information should tell what the job is doing, and provide a limited possibility of interaction with it to allow publication of specific data. Three types of publication will be defined:

1. Any Gaudi job will publish a default set of parameters that the user or production manager might want to inspect, such as the configuration of the application (version number, name of executable, detector database version), various runtime parameters (channel type, qq user decay file, geant cards) and so on. These will be published twice (at the start and end of the job) and as often as required by the subscriber. If there is no subscriber, the publication process will not interfere with the normal running of the job. Dynamic information such as the event number currently being processed, as well as warnings and error messages from the Gaudi message service are also published by default.
2. It will be possible to send commands to a running job from the monitoring user interface. These commands will allow the publication of any required '*name*, *value*' pair. This could enable the inspection of on-line filling of histograms for instance.
3. All environment variables that are defined by the script that contains the Gaudi executable are exposed to the monitoring system. The user can also define checkpoints with variable messages and error codes in the scripts.

The published information will be stored in a (transient) control database. Upon completion of the job, the data in the production database is updated and the data erased from the control system.

A complication arises because messages will be sent to and from jobs through IP communication ports – for security reasons it is not always possible to open ports for sending and receiving in a running job. This will impose some constraints on the architecture.

7.2 Definitions

Monitoring system: A user interface that receives and displays information that is published by running applications. A possibility for limited interaction exists by sending commands to running jobs.

Control system: A stand-alone system which is used to manage the communication via messages to and from the running jobs. The control system has a database. Memory space is available to communicate files to the monitoring UserInterface.

Production database: the database that contains all unfinished requests for data production. A demon subscribes to the control system so that when a job finishes, its entry in the production database is updated. The entry is then removed from the controls database.

Job script generator and job submission program: this tool is described in the previous chapter.

7.3 Interaction with the service

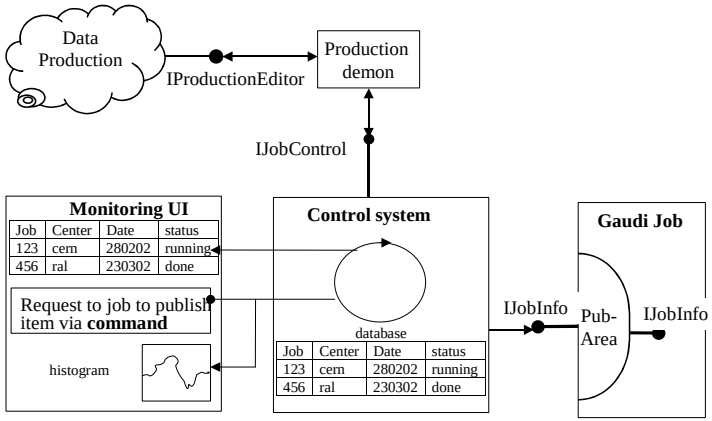

Figure 30 A sketch of the job control and submission system. The production demon submits jobs as the production area requests them. The demon makes use of the controls system, which also allows top monitor data items published by the Gaudi job. Monitoring utilities in addition are able to request data items from the Gaudi job.

The control system database is organized on a 'per job' basis.

The following data must be accessible through the control system:

- Configuration parameters of the running job
- Details of the running job (where, when)
- Meta data of files being produced
- Dynamic parameters of running jobs (current event number)
- Any data publishable as (name, value) pairs
- Bookkeeping data for the production database

Where:

- The running Gaudi job and the job script must provide information.
- The monitoring UI subscribes to information from the control system
- The monitoring UI allows to send a job a request for publication of a (name, value) pair
- The control system publishes bookkeeping data to the production database

Gaudi jobs implement the following functionality:

- **Active publishing:** Publish typed information (name, item) - to expose a set of default objects to the control system. This information is updated whenever the job itself re-publishes the data item. Published data items are cached by the control system, the monitoring facility does not have to interrogate the Gaudi job, it is sufficient to interact with the control system.
- Allow access to job objects (name, item, size) if the job objects can be serialized and relocated. This allows to access snapshots of existing objects. The objects must be polled from the Gaudi job by the monitoring UI.
- By default, data is only published at the start and end of the job, except if a client (the monitor) subscribes on a regular basis

The control system stores the information actively published by the Gaudi job in its database

- A production demon updates the production database upon completion of the job
- Passes information to the monitoring UI

The monitoring UI displays the default information stored in the control system

- Activates a subscribe to dynamic information (e.g. current event number)
- Allows retrieval of objects from running jobs and to display the retrieved objects such as histograms from the histogram data store.

The production demon

- Submits new jobs and controls execution jobs using the information stored in the production database.
- Updates the production database with predefined and always published data items required by this domain such as the current event number.

This architecture should be independent of any underlying (Grid-) technology. However, a feasible (and partially existing) implementation is based on PVSS for the monitoring UI and the control system, and Dim (http://dim.web.cern.ch/dim) for the transmission of messages between the running job and PVSS. Clearly, such an implementation offers additional functionality such as automatic updates on data change etc., which are not reflected by generic interfaces.

7.4 Interface Model

The interfaces provided by the job control and monitoring system must support two activities:

- The components of the Gaudi job must be able to publish data items and must be able to accept requests to access objects. This is handled by the **IJobInfo** interface as it is shown in Figure 31. Once published, these data items must be accessible by the control system – through the same interface. Hence, this interface must allow to publish data items identified by a name
- The controls system allows controlling the execution of jobs accessing the information offered by the Gaudi job through the **IJobControl** interface as it is shown in Figure 32. Such control activities include the
  - Job submission
  - Job cancellation
  - Set jobs in hold/resume
  - Retrieve published data items identified by their name
  - Retrieve (serialized) objects identified by their name

These interfaces are implemented by two different components – one used internally by the Gaudi job and another one used and offered by the controls system, hence a possible implementation will result in two services as shown in

Figure 33.

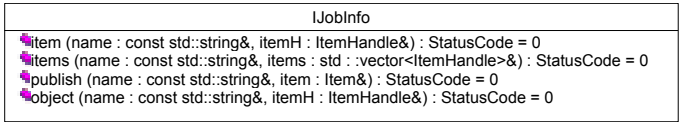

Figure 31 The IJobInfo interface.

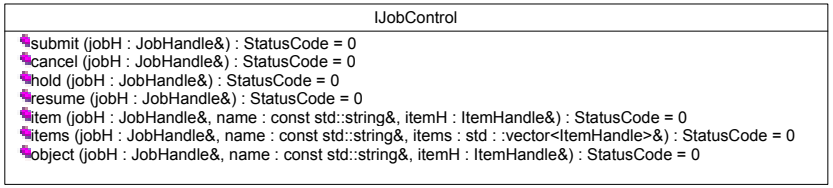

Figure 32 The IJobControl interface.

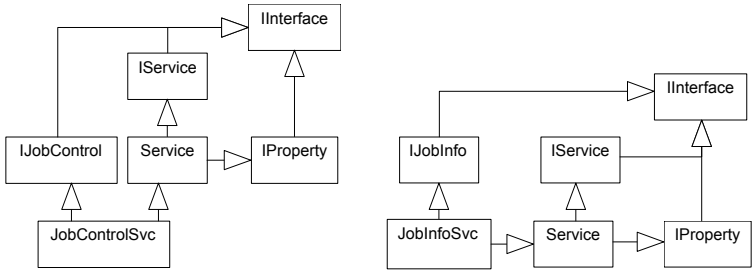

Figure 33 A possible implementation of the components, which offer the interfaces is described in Figure 31 and Figure 32.

8 Deliverables

The goal of this project is the deliver a possible implementation of all tools, which allows

1. The management of versioned configurations of Gaudi applications in terms of application options and parameters (Job Configuration)
2. The control and management of data processing activities such as Monte-Carlo productions.
3. The management and access to the files produced during these data management activities.

This functionality will be implemented as a set of Gaudi services, which can be accessed using abstract interfaces. Using these interfaces the basic tools can easily be implemented as a command line callable wrapper around these interfaces. The project will hence deliver

- A C++ callable API for the implementation of the interfaces described in section 3.5. The API implementation, which will be used by all tools, has highest priority.

Job Configuration

- A tool to create/edit job configurations. “One configuration” consists out of all job options needed by a single application such as Brunel and all required parameters to build a script executing the application.
- A tool to extract job options as a Gaudi job options file from the job configuration for a given application identified by its name and its configuration version.
- A tool to generate a standard script used to execute an application identified by its name and its configuration version from the job configuration.

Bookkeeping

- A tool to publish files in the bookkeeping database.
- A World Wide Web based interface to access files using a standard web browser. The functionality of the initial interface should cover existing functionality.

Job Submission

- A controls system, which allows to submit/control jobs and retrieve information from existing jobs.
- A tool to visualize the status of executing production jobs. Depending on the technology used this involves the development of additional components for the controls system.

Data Production

- A tool to edit data production workflows.
- A tool to instantiate execution flows as the result of a data processing request to the data production.
- The daemon responsible for creating jobs from the information stored in the data production and job configuration database, which submits jobs to a data processing facility.

All tools should cover the managerial work described in 1-3. Although the tools will be rather primitive in the beginning, they should allow all necessary actions. Later, once the API is stabilized more sophisticated tools involving GUI designs are envisaged.