

GANGA: a user-Grid interface for Atlas and LHCb

K. Harrison

Cavendish Laboratory, University of Cambridge, CB3 0HE, UK

W. T. L. P. Lavrijsen, C. E. Tull

LBNL, Berkeley, CA 94720, USA

P. Mato

CERN, CH-1211 Geneva 23, Switzerland

A. Soroko

Department of Physics, University of Oxford, OX1 3RH, UK

C. L. Tan

School of Physics and Astronomy, University of Birmingham, B15 2TT, UK

N. Brook

H.H. Wills Physics Laboratory, University of Bristol, BS8 1TL, UK

R. W. L. Jones

Department of Physics, University of Lancaster, LA1 4YB, UK

GANGA is a user interface that gives access both to local resources and to the Grid, and provides job-configuration and data-management tools matched to the computing environments of the particle-physics experiments Atlas and LHCb. It is being developed in python, following a component architecture, with components interacting via a so-called software bus. Components of general applicability deal with tasks such as workflow definition, script generation, job submission, file transfer to and from remote sites, and monitoring. This core functionality can be supplemented as needed by components tailored to meet the requirements of specific groups of users, so that GANGA is readily extensible. Specialised components for Atlas and LHCb incorporate knowledge of the experiments' common software framework and configuration-management system, and simplify tasks such as application configuration, job splitting, the setting-up of the run-time environment, and output collection. Further components, specialised for one or the other of Atlas and LHCb, allow access to various experiment-specific databases.

This paper gives details of the GANGA design and implementation, the development of the underlying software bus architecture, and the functionality of the first public GANGA release.

1. INTRODUCTION

The Atlas [1] and LHCb [2] experiments will study the products of high-energy proton-proton interactions at the Large Hadron Collider (LHC) [3] currently under construction at CERN, Geneva, with startup scheduled for 2007. Both experiments involve many hundreds of physicists, from tens of institutes, and will require analysis of data volumes of the order of petabytes per year. The experimental data, recorded at CERN, will be shared between sites for the processing required to extract information on particle trajectories and identities (event reconstruction). Data for the large samples of simulated interactions that must be generated to achieve a full understanding of detector behaviour and of the physics effects of interest will also be distributed between multiple locations. Grid services will be exploited to allow the participating physicists transparent access to the globally distributed datasets, and to the decentralised computing resources available to each experiment.

The GAUDI/ATHENA [4, 5] software framework used by LHCb and Atlas, usually referred to hereafter simply as GAUDI, is designed to support the full spectrum of data-processing

applications, including simulation, reconstruction, and physics analysis. A joint project has been set up to develop a front-end that will aid in the handling of framework-based jobs, and performs the tasks necessary to run these jobs either locally or on the Grid. This front-end is the GAUDI/ATHENA and Grid Alliance, or GANGA [6].

GANGA covers all phases of a job's life cycle: creation, configuration, splitting and re-assembly, script generation, file transfer to and from worker nodes, submission, run-time setup, monitoring, and reporting. In the specific case of GAUDI jobs, the job configuration includes selection of the algorithms to be run, definition of algorithm properties, and specification of inputs and outputs. GANGA relies on middleware from other projects, such as Globus [7] and EDG [8], to perform Grid-based operations, but makes use of the middleware functionality transparent to the GANGA user.

In this paper, we present the GANGA design and the choices made for its implementation. We report on the work done in implementing the software bus that is a key feature of the design, and we describe the functionality of the current GANGA release.

2. GANGA DESIGN

2.1. Overview

GANGA is being implemented in python [9], an interpreted scripting language, using an object-oriented approach, and following a component architecture. The python programming language is simple to use, supports object-oriented programming, and is portable. By virtue of the possibilities it allows for extending and embedding, python is also effective as a software “glue.” A standard python installation comes with an interactive interpreter, an Integrated Development Environment (IDE), and a large set of ready-to-use modules. The implementation of the component architecture underlying the GANGA design benefits greatly from python’s support for modular software development. The components of GANGA interact with one another through, and are managed by, a so-called software bus [10], a prototype of which is described in Section 3. This scheme is represented graphically in Fig. 1.

As considered here, a component is a unit of software that can be connected to, or detached from, the overall system, and brings with it a discrete, well-defined, and circumscribed functionality. In practical terms, it is a python module (either written in python or embedded) that follows a few non-intrusive conventions. The component-based approach has the advantages that it allows two or more developers to work in parallel on well-separated tasks, and that it allows reuse of components from other systems that are architecturally similar. The functionality of the GANGA components can be accessed through a Command-Line Interface (CLI), and through a Graphical User Interface (GUI), built on a common Application-Programmer Interface (API). All actions performed by the user through the GUI can be invoked through the CLI, allowing capture of a GUI session in an editable CLI script.

The components can be divided into three categories: general, domain specific, and external. Further developments will add components as needed. Currently defined components are discussed below, with reference to Fig. 1.

2.2. General components

Although the first priority is to deal with GAUDI jobs, GANGA has a set of core components suitable for job-handling tasks in a wide range of application areas. These components are shown to the right of the software bus in Fig. 1.

The cornerstone of the system is a job-definition component, which characterises a GANGA job in terms of the following:

- The name chosen as the job’s identifier.
- The workflow, discussed below, which indicates the operations to be performed when the job is run.
- The computing resources required for the job to run to completion.
- The job status (in preparation, submitted, running, completed).

A job workflow is represented as a sequence of elements (executables, parameters, input/output files, and so on), with the action to be performed by, and on, each element implicitly defined. Resources required to run a job, for example minimum CPU time or minimum memory size, are specified as a list of attribute-value pairs, using a syntax not tied to any particular computing system. The job-definition component implements a job class, and classes corresponding to various workflow elements.

Other GANGA components of general applicability perform operations on, for, or using job objects:

- A job-registry component allows for the storage and recovery of information for job objects, and allows for job objects to be serialised.
- A script-generation component translates a job’s workflow into the set of (python) instructions to be executed when the job is run.
- A job-submission component takes care of submitting the workflow script to a destination indicated by the user, creating Job Description Language (JDL) files where necessary, and translating the resource requests into the format expected by the target system (European Data-Grid (EDG), GridPP Grid, US-ATLAS Testbed, local PBS queue, and so on).
- A file-transfer component handles transfers between sites of job input and output files, this usually involving the addition of appropriate commands to the workflow script at the time of job submission.
- A job monitoring component keeps track of job status, and allows for user-initiated and scheduled queries.

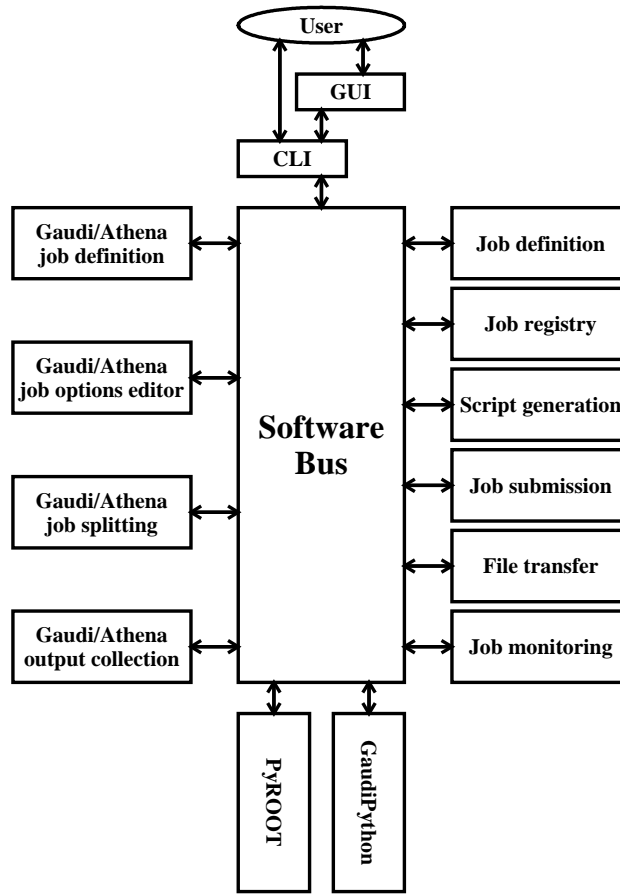


Figure 1: Schematic representation of the GANGA design, which is based on components interacting via a software bus. The user issues commands either via the graphical user interface (GUI) or via the command-line interface (CLI). GANGA components of general applicability are shown on the right side of the software bus, whereas GANGA components dedicated to specific requirements of the GAUDI framework are shown on the left. Components external to GANGA are shown at the bottom: GAUDIPYTHON and PYROOT are python interfaces to GAUDI and ROOT respectively.

2.3. Domain-specific components

GANGA is optimised for a given problem domain through the addition of application-specific components. For the current user groups, Atlas and LHCb, components shown to the left of the software bus in Fig. 1 incorporate knowledge of the GAUDI framework:

- A component for GAUDI job definition adds classes for workflow elements not dealt with by the general-purpose job-definition component. For example, applications based on GAUDI are packaged using a configuration management tool (CMT) [11], which requires its own workflow elements. Also, this component provides workflow templates covering a variety of common tasks, such as simulating events and analysing some dataset.
- A component for GAUDI job-options editing allows selection of the algorithms to

be run and modification of algorithm properties.

- A component for GAUDI job splitting allows for large jobs to be broken down into smaller sub-jobs, for example by examining the list of input data files and creating jobs for subsets of these files.

- A component for GAUDI output collection merges outputs from sub-jobs where this is possible, for example when the output files contain data sets in the form of histograms or ntuples.

Specialised components for other application areas can be readily added. The subdivision into general and specialised components, and the grouping together of specialised components dedicated to a particular problem domain, allows new user groups to identify quickly the components that match their needs, and will help ensure GANGA stability.

2.4. External components

The functionality of the components developed specifically for GANGA is supplemented by the functionality of external components. These include all modules of the python standard library, and also non-python components for which an appropriate interface has been written. Two components of note in the latter category, both interfaced using BOOST [12], allow access to the services of the GAUDI framework itself, and to the full functionality of the ROOT analysis framework [13].

3. PYTHON SOFTWARE BUS

3.1. Functionality

To first order, the software bus functionality required by GANGA is provided by the python interpreter itself. In particular, this allows module loading and method-call binding to be performed dynamically, and has support for managing component lifecycles. The main features not offered by the interpreter, but nevertheless desirable are:

- **Symbolic component names**

Python modules are loaded on the basis of their names, which are mapped one-to-one onto names in the file system, sometimes including (part of) the directory structure. It should, in addition, be possible to load components on the basis of the functionality that they promise.

- **Replacing a connected component**

This is different from the standard reload functionality, which loads a new version of a current module and doesn't rebind any outstanding references. A component may, however, need to be completely replaced by another component, meaning that the latter must be reloaded at a deeper level, and references into the old component must be replaced by equivalent references into the new component, wherever possible.

- **Disconnecting components**

A standard python module is not unloaded until all outstanding references disappear. This is common behavior in many off-the-shelf component architectures. However, it should be possible to propagate the unloading of a component through the whole system, allowing for more natural interactive use.

- **Configuration and dependencies**

Since python modules simply execute python code, their configuration and dependencies are usually resolved locally. Components should be able to advertise their configurable parameters and their dependencies, such that it is also possible to manage the configuration externally and/or globally.

The software bus should also support a User Interface Presentation Layer (UIPL), through which the configuration, input and output parameters, and component functionality can be connected to user interface elements. The bus inspects the component for presentable elements, including (if applicable) their type, range, name, and documentation. It subsequently requests the user interface to supply elements that are capable of providing a display of and/or interaction with each of the parameters, based on their type, range, and so on. Both the interface element and the component should then be connected through the UIPL.

3.2. The PyBus prototype

A prototype of a software bus (PYBUS) has been written to explore the possibilities for implementing the above features in a user-level python module, rather than in a framework. That is, PYBUS is a client of the python interpreter and does not have any privileges over other modules. This means that components written for PYBUS will act as components when used in conjunction with the bus, or as python modules when used without. Conversely, python modules that were not written as PYBUS components can still be loaded and managed by the software bus, assuming that they adhere to standard python conventions concerning privacy and dependencies.

A user connects a component to PYBUS, and so makes it available to the system, using the component's logical, functional, or actual name. Components available to PYBUS must be registered under logical names, optionally advertising under functional names the public contracts that they fulfill. If a component is not registered, it can only be connected using its actual name, which is the name that would be used in the standard way of identifying a python module. Unlike the actual name, which has to be unique, the logical name and functional names may be claimed by more than one component. PYBUS chooses among the available components on the basis of its own configuration, a priority scheme, or user action.

In the process of connecting a module, PYBUS looks for parameters starting with “Py-Bus_” in the dictionary of the module in which the component is contained. These parameters may describe dependencies, new components to be registered, post-connect or pre-disconnect configuration, and so on. It is optional for a module to provide such parameters and PYBUS will use some heuristics if they are absent. For example, all public identifiers are considered part of the interface, so that any module can be connected as if it were a component. The user has free choice over the name under which the module should be connected, the default being the logical name.

PYBUS allows component replacement, using the garbage collection and reference-counting information of the python interpreter to track down any outstanding references, and acting accordingly. Some references, for example those to variables or instances, are rebound, whereas others, for example object instances, are destroyed. Disconnecting a component is rather similar to replacing it, except that no references are rebound: all are destroyed.

Python allows user modules to intercept the importing of other modules, by replacing the import hook. This mechanism allows the PYBUS implementation to bookmark modules that are imported during the process of connecting a component, and consequently to manage component dependencies.

The implementation of the prototype software bus has been mostly successful. There are a few issues still to be resolved for embedded components, but for pure python components it has been shown that it is possible to implement the component architecture features missing in the python interpreter with a user-level python module.

4. FIRST RELEASE

4.1. Overview of functionality

The GANGA design, including possibilities for creating, saving, modifying, submitting and monitoring jobs, has been partly implemented, and has been released for user evaluation. The tools implemented are suitable for a wide range of tasks, but we have initially focused on running one type of job for Atlas and one type of job for LHCb, focusing on the ATLFAST [14] fast simulation in the case of the former, and the DAVINCI [15] analysis in the case of the latter. Optimisation for other types of applications essentially means creating more templates, which

is a straightforward procedure. Communication between components in the prototype is via the python interpreter, with the sophistication of PYBUS to be added later.

The current release implements only a part of the intended functionality, but already simplifies a number of tasks. For example, the creation of the JDL files necessary for job submission to the EDG Testbed, and the generation of scripts to submit jobs to other batch systems, has been automated.

Most parameters relevant for ATLFAST and DAVINCI jobs have been given reasonable default values in GANGA, so that a user only has to supply minimal information to create and configure a new job. Existing jobs can easily be copied, renamed, and deleted. When a job is created, it is presented as a template that can be edited by the user. After making the required modifications, the user can submit the job for execution with a single command.

A generic job-splitting mechanism has been implemented, where a splitter function is used to specify the way in which the required sub-jobs differ from a single initial job set up by the user. For the production of simulated interactions, for example, the splitter function might return the values of the random-number seeds for each sub-job.

When a job is submitted, GANGA starts to monitor the job state by periodically querying the appropriate computing system. This process can be stopped or started manually at any time.

When a job completes, the output is automatically transferred to a dedicated directory, or to any other location specified for the job output files.

Below, we give details of GANGA’s current job-handling capabilities and the main graphics features: the GUI and the job-options editor.

4.2. Job-handling capabilities

The job registry, introduced in Section 2.2, catalogues information (metadata) for all user jobs. It also acts as control centre for channelling job-related commands, which deal with creation, configuration, submission, termination, and monitoring.

As detailed in Section 2.2, a user job is represented in GANGA as an object that includes information about the job status, the associated workflow, and the computing resources required. The specific steps for job submission, and monitoring, which differ for different types of computing system, are delegated to a so-called job handler. For submission, the

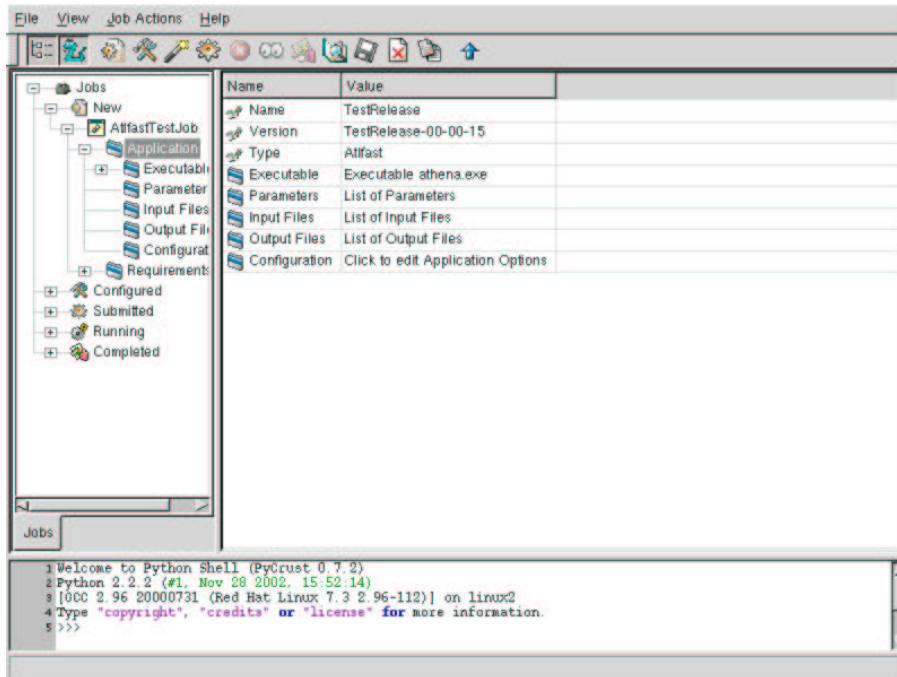


Figure 2: Screenshot showing the general layout of the Ganga GUI.

job handler parses the job resource requirements and translates to system-specific commands. For monitoring, the job handler returns system-dependent information about job status. Ganga currently has components containing job handlers to work with the local computer, a local PBS or LSF batch system, or the EDG Testbed.

In the first release of Ganga, a single, generic workflow is implemented. This allows specification of the application to be run (executable or script), configuration parameters, and input and output files. For transferring the input and output files to and from worker nodes, Ganga makes use of the local system copy command, the gridftp transfer protocol, and the EDG sandbox mechanism. The transfer method is set up automatically by the job handler prior to job submission.

4.3. Graphical user interface

The Ganga GUI is based on WXPYTHON, the extension module that embeds the WXWINDOWS platform-independent application framework. WXWINDOWS is implemented in C++, and is a layer on top of the native operating and windowing system. The design of the GUI is based on a mapping of major Ganga core classes – jobs, executables, files, and so on – onto the corresponding GUI classes. These classes (GUI handlers) add the hooks necessary to provide interaction with the graphical

elements of the interface, on top of the functionality of the underlying core classes.

The basic display of the Ganga GUI is shown in Fig. 2. The main window is divided into three parts: the left section displays the job tree; the right section is used to display a variety of panels where user input is given, for example for job setup (Fig. 2) and for defining sequences (Fig. 3); the lower section hosts an embedded python shell (PYCRUST, itself designed for use with WXPYTHON), and doubles as a log window.

With the expert view enabled, the hierarchy of all job-related values and parameters is shown in the job tree. The most important values are brought to the top of the tree; less important ones are hidden in the branches. The normal (user) view stops at the level of jobs and gives access to the most important parameters only. A control list displays the contents of the selected folder in the job tree. Most values in the list can be selected for editing through a double mouse click.

Actions on jobs can be performed through a menu, using a toolbar, or via pop-up menus called by a right click in various locations.

When the monitoring service is switched on, jobs move automatically from one folder to another as their status changes. To avoid delays in the program response to user input, the monitoring service runs on its own thread. It posts customised events whenever the state of a job changes and the display is to be up-

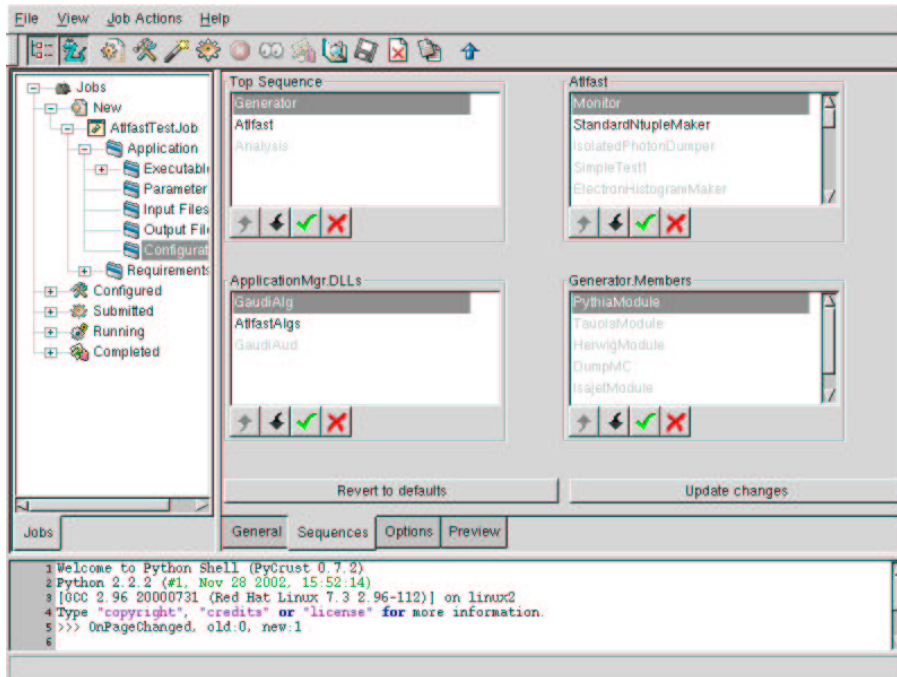


Figure 3: Screenshot of GANGA, showing one of the windows presented by the job-options editor. The example window is for defining sequences.

dated. For GUI updates not related to job monitoring, GANGA handles the standard update events posted by the WXWINDOWS framework.

4.4. Job-options editor

The job-options editor (JOE) has been developed in the context of work on ATLFAST, and allows the user to customise ATLFAST jobs from within the GANGA environment. The next step will be to generalise JOE, so that it may be used in association with any GAUDI job.

The current main features of JOE are as follows:

- Through its GUI (Fig. 3), JOE assists the user in configuring job options from inside GANGA, eliminating the possibility of errors arising from incorrectly spelt option names and incorrect syntax.

The user can define sequences of algorithms and select dynamic libraries by enabling or disabling entries and arranging them in a desired order.

JOE incorporates an option-type aware presenter, which chooses the correct presentation format at run-time for individual job options based on their attributes (for example, drop-down menus

for discrete-choice options, arbitrary-value entry for simple-choice options, value appending for list-type options).

- Job-option settings for commonly performed jobs can be stored and reused. This saves the user the work of re-entering option values for subsequent jobs, especially if only minor modifications are needed.
- Once all the options have been set, the preview function allows the user to check that the created script is as required.
- Following the basic GANGA philosophy, all functionality of the editor is also available on the python command line without the GUI. Users and developers alike can make use of this API.

In addition to the generalisation to all types of GAUDI job, several other improvements to JOE are foreseen:

- Option attributes that enable JOE to choose appropriate presentation formats for individual job options are currently hard-coded. Future versions will attempt to make deductions about job option attributes at run-time.
- Editable previews of options files will allow the user to make last-minute changes.

- GAUDI jobs may be characterised by several hundred job options. It will not be useful to display all options indiscriminately, and so some form of information hiding is required.

A “favourite-options first” feature will further speed up the user’s task of job-option modification, by placing frequently used options at the top of the list, perhaps hiding options used less often.

- Although rudimentary option-value checks are performed, the more important range checking is not yet available. This feature requires permitted ranges to be defined as attributes of individual options.

JOE showcases the extensibility of the GANGA user interface. Future extensions can be developed and incorporated in the same way.

5. OUTLOOK

The first release of the GANGA package has been made available. Test users from Atlas and LHCb have successfully submitted jobs through GANGA, and have given positive feedback. The development schedule laid out for the remainder of calendar year 2003 is targeted at providing a product to satisfy requirements for the Atlas and LHCb data challenges, and associated physics analyses. Cooperation and integration with existing projects (ASK [16], ATCOM [17], DIRAC [18], DIAL [19]) is foreseen in order to meet in time these requirements. The GANGA project will then continue to keep pace with, and adapt to, the ever-evolving Grid middleware services.

Acknowledgments

This work was partly supported through the UK GridPP project; and by the Office of Science, High Energy Physics, U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

References

- [1] Atlas Collaboration, “Atlas - Technical Proposal”, CERN/LHCC94-43, CERN, December 1994.
- [2] LHCb Collaboration, “LHCb - Technical Proposal”, CERN/LHCC98-4, CERN, February 1998.
- [3] LHC Study Group, “The LHC conceptual design report”, CERN/AC/95-05, CERN, October 1995.
- [4] P. Mato, “GAUDI - Architecture design document”, LCHb-98-064, CERN, November 1998;
<http://cern.ch/lhcb-comp/Frameworks/Gaudi>
- [5] <http://atlas.web.cern.ch/Atlas/GROUPS/SOFTWARE/00/architecture/General>
- [6] <http://ganga.web.cern.ch/ganga>
- [7] <http://www.globus.org>
- [8] <http://cern.ch/eu-datagrid>
- [9] G. van Rossum and F. L. Drake, Jr. (eds.), “Python Reference Manual”, Release 2.2.2, PythonLabs, October 2002;
<http://www.python.org>
- [10] P. Mato et al., “SEAL: Common core libraries and services for LHC applications”, these proceedings.
<http://www.cmts.site.org>
- [11] <http://www.cmts.site.org>
- [12] <http://www.boost.org>
- [13] R. Brun, F. Rademakers, and S. Panacek, “ROOT, an object-oriented data analysis framework”, CERN/2000-013, CERN, 2000;
<http://root.cern.ch>
- [14] E. Richter-Was, D. Froidevaux, and L. Poggioli, “ATLFAST 2.0 a fast simulation package for ATLAS”, ATLPHYS-98-132, CERN, 1998;
<http://www.hep.ucl.ac.uk/atlas/atlfast/>
- [15] <http://cern.ch/lhcb-comp/Analysis/>
- [16] W. T. L. P. Lavrijsen, “The Athena Startup Kit”, Proc. 2003 Conference for Computing in High Energy and Nuclear Physics, La Jolla, California, USA
- [17] V. Bertin, L. Goossens, C. L. Tan, “ATLAS Commander: an ATLAS production tool”, Proc. 2003 Conference for Computing in High Energy and Nuclear Physics, La Jolla, California, USA
- [18] A. Tsaregorodtsev et al., “DIRAC - Distributed Implementation with Remote agent Control”, Proc. 2003 Conference for Computing in High Energy and Nuclear Physics, La Jolla, California, USA
- [19] D. Adams et al., “DIAL - Distributed Interactive Analysis of Large datasets”, Proc. 2003 Conference for Computing in High Energy and Nuclear Physics, La Jolla, California, USA