LHCb-2004-089

October 15, 2004

# PYTHON-BASED PHYSICS ANALYSIS ENVIRONMENT FOR LHCb

I. Belyaev, LAPP, Annecy-le-Vieux, France
M. Frank, P. Mato, CERN, Geneva, Switzerland
G. Barrand, LAL, Orsay, France
A. Tsaregorodtsev, U. Marseille II, Marseille, France
E. de Oliveira, CBPF, Rio de Janeiro, Brasil

*Abstract*

BENDER is the PYTHON based physics analysis application for LHCb. It combines the best features of the underlying GAUDI software architecture with the flexibility of the PYTHON scripting language and provides end-users with a friendly physics analysis oriented environment.

## INTRODUCTION

The objective is to provide LHCb [1] physicists with a friendly environment for doing physics analysis on reconstructed event objects (ESD) as well as physics analysis objects (AOD) and event tags, and to be able obtain the best physics results. The main driving qualities for this new physics analysis environment, BENDER, have been:

- *Rapid Application Development (RAD) paradigm*. This is to enable physicists to try out new analysis ideas by developing fast prototypes.
- *Interactivity*. The possibility to (re)define the algorithms, parameters and configuration in the process of code development from the interactive program prompt.
- *Readability* and *Compactness*. The end-user physics analysis code should be understandable by colleagues, unambiguously matching the physical description and with minimal pollution from technical details.
- *Completeness*. This is the usability of the environment for developing real analysis.
- *Integratibility*. Ease of integration with third party products, in particular the software widely used within HEP community (e.g. for visualization and statistical analysis).

BENDER is based on GAUDI [2, 3, 4] event-processing software framework. *Component*s in the GAUDI framework implement a number of abstract interfaces (pure abstract classes in C++) for interaction with other *Component*s. Among the major generic categories of *component*s one finds *Algorithm*s, *Service*s and *Tool*s. *Service*s and *Tool*s are categories of components which offer services directly or indirectly needed by *Algorithm*s. The GAUDI framework is used for building all LHCb event data processing applications: GAUSS [5] for simulation, BOOLE for digitization, BRUNEL for reconstruction, PANORAMIX [6] for visualization and event display and DAVINCI [7] for physics analysis,

## ARCHITECTURE AND FRAMEWORK

### General Overview

BENDER is based on the generic PYTHON [8] bindings for the GAUDI framework, called GAUDIPYTHON, and on a high-level C++ physics analysis toolkit called LOKI [9]. LOKI in turn uses the *Tool*s and *Algorithm*s developed in C++ in the context of the DAVINCI framework. The schematic relations between the DAVINCI and BENDER frameworks are sketched in Figure 1.

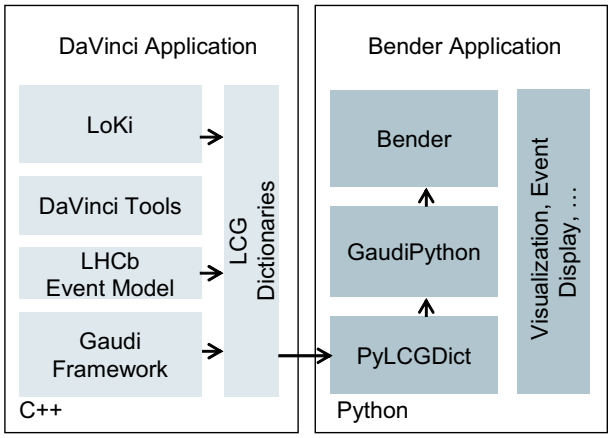


Figure 1: The schematic relations between the DAVINCI and BENDER frameworks

The interactivity of BENDER is provided via the usage of PYTHON, which is a high-level object oriented scripting language with support for C/C++ extensions. The RAD paradigm is one of the most essential features of PYTHON. BENDER is mainly implemented as set of PYTHON extension modules.

The usage of PYTHON, AIDA [10] abstract interfaces and standard LCG [11] reflection techniques [12] allows an easy integration of BENDER's analysis environment with

LHCb-PROC-2004-021
30/09/2004

third party products like the interactive event display and visualization tools like PANORAMIX, ROOT [13] and HIPPODRAW [14].

### *C++ Analysis Toolkit*

LOKI is a C++ toolkit for Physics Analysis that provides a set of high level analysis utilities with physics oriented semantics. The package has been inspired by the great success of the KAL program, used for physics analysis by the ARGUS collaboration, and the PATTERN [15] package, used by the HERA-B collaboration. The ideas from GCOMBINER [16], LOKI [17] and CLHEP [18] libraries are also used. The current functionality of the package includes

- Set of predefined *Functors* and generic operations
- Selection and filtering of particles and vertices,
- Multi-particle combinatory loops
- Intuitive interface to histograms and N-Tuples, in particular it includes book-and-fill on demand idiom.
- Simple matching of reconstructed objects with Monte Carlo truth information

The end-user code is not polluted by technical details and a clear separation between physics analysis code and technical details is achieved as illustrated in Figure 2.

From a formal point of view the physicist inherits his analysis algorithm from the base class `LoKi::Algo` and implements the virtual method `LoKi::Algo::analyse`.

The majority of complicated physics analysis idioms can be expressed by only one line of LOKI code. It has been demonstrated that usage of LOKI results in a drastic reduction of the number of lines of code. In order to make the end-user code even more compact, the concepts of *Pattern*s and implicit loops in the spirit of STL algorithms have been introduced, e.g. the explicit loops from the example in Figure 2 could be substituted by the following implicit loop:

```
pattern ( "D0" , "K- pi+" , "D0" ,
  ADMASS( "D0" ) < 30 * MeV , VCHI2 < 4 );
plot    ( M - M1 , loop( "D0 pi+" ) ,
          "DM for D*+" , 130 , 180 );
```

LOKI-based analysis code is further enhanced by the concept of *locality*, in which the entities are declared and defined only at the place they are used. The "book-on-demand" treatment of histograms and N-Tuples illustrates this important concept.

There are no raw C++ pointer manipulations and explicit memory management in LOKI-based physics analysis code. This fact together with the suppression of explicit and tedious loops makes the code more error-safe and easy to debug.

The implementation of LOKI heavily exploits the modern technique of generic template meta-programming [17, 19]. In general, LOKI code is very efficient due to the templated nature and the fact that most of the code is in-lined. The kernel components of LOKI are loosely coupled with LHCb Event Model.

### *Python Binding*

For the C++/PYTHON binding, the LCG reflection technique [12] of dynamic binding developed by the SEAL [20] project has been chosen. With this approach, C++ objects are accessible from PYTHON through *dictionaries*.

*Dictionaries* are produced only for the major GAUDI interfaces (GAUDIPYTHON package) and basic LOKI classes (BENDER package) as well as for LHCb Event Model classes, which are needed for object persistency. The existing *dictionaries* for CLHEP, AIDA and ROOT classes are available through the SEAL and PI [21] projects.

### *Job and Application Configuration*

To complete the analysis environment we need to provide easy ways to end-users to configure and set-up the running environment. This requires selecting the required version of the software packages and to configure the analysis application itself. The job configuration and environment setup is performed using a PYTHON module `cmt.py` that interfaces to the CMT[22, 23] tool. A typical set of job configuration lines is:

```
>>> import cmt
>>> cmt.project( 'Bender' , 'v4r0' )
>>> cmt.use( package = 'Ex/BenderExample' )
>>> cmt.setup()
```

The application configuration, which implies in particular the scheduling of *Algorithm*s and the configuration of all *Components* is performed by the PYTHON module `gaudimodule.py` from GAUDIPYTHON package:

```
>>> import gaudimodule
>>> g = gaudimodule.AppMgr()
>>> g.TopAlg += [ 'MyAlg1' , 'MyAlg2']
>>> g.OutputLevel = Info
>>> g.run(100)
```

### *Physics Analysis Components*

For development of the basic physics analysis algorithms in PYTHON it is sufficient to have access to LHCb Event Model classes, generic GAUDI services, physics analysis *Tool*s and *Algorithm*s from the DAVINCI framework. Such an approach requires a direct manipulation of Event Model objects, multiple nested loops, calling the analysis *Tool*s directly, etc. This mode will inevitably add a significant PYTHON-related CPU penalty.

The above approach is available for BENDER, but the heart of the BENDER framework, the `bendermodule.py` is built as a PYTHON extention module around the *Dictionaries* constructed for classes and utilities from LOKI Toolkit. The bulk of actual computation is performed in C++ by LOKI utilities, therefore one gets only a minimal penalty from usage of PYTHON.

Almost all functionality, offered in C++ by LOKI is available in PYTHON via the `bendermodule`. Where possible,

```
#include "LoKi/LoKi.h"
LOKI_ALGORITM( Dstar )
{
  select ( "K-"  , "K-"  == ID && PT > 1 * GeV ) ;
  select ( "pi+" , "pi+" == ID && P  > 3 * GeV ) ;

  Cut masscut = abs( DMASS ( "D0" ) ) < 30 * MeV ;
  for ( Loop D0 = loop ( "K- pi+" , "D0" ) ; D0 ; ++D0 )
  {
    if ( VCHI2 ( D0 ) < 4 && masscut ( D0 ) ) { D0 -> save ( "D0" ) ; }
  }
  for ( Loop Dst = loop ( "D0 pi+" ) ; Dst ; ++Dst )
  {
    plot ( ( M ( Dst ) - M1 ( Dst ) ) / MeV , "DM for D*+"   , 130 , 180 ) ;
  }
  return StatusCode::SUCCESS ;
};
```

Figure 2: Simple example of analysis code for the "selection" of $D^{*+} \to D^0\pi^+$, followed by $D^0 \to K^-\pi^+$ using the LOKI Toolkit.

the PYTHON semantics with keyword arguments is added in addition to the semantics of positional argument, available from the *Dictionaries*. Similar to the case of LOKI, a physicist inherits his analysis algorithm from the base class `Algo` and implements the method `Algo::analyse`. The normal services are available to the physicist algorithm. For example one can use N-Tuples in the following way:

```
tup = self.nTuple( title='My D*+ N-Tuple' )
tup.column( name='M'  , value = M (Dst) )
tup.column( name='M1' , value = M1(Dst) )
tup.column( name='P'  , value = P (Dst) )
tup.column( name='PT' , value = PT(Dst) )
tup.write()
```

The complete example is presented in Figure 3.

One can compare the `C++` code, written using LOKI Toolkit with similar PYTHON code, based on BENDER. One sees that the semantics of LOKI and BENDER are very similar and conversion is possible. For example, for the development of the algorithm and tuning of selection cuts one can use BENDER with the subsequent conversion to `LoKi` code after convergence of the iterative development cycle.

The hybrid approach, called LOKIHYBRID has also been developed, which allows us to use BENDER code fragments (e.g. selection *cuts*) from the `C++` algorithm with no PYTHON-related CPU run-time overhead.

### *Visualization*

Histograms filled in BENDER can be visualized through their abstract AIDA interface in the same interactive BENDER session.

A few external tools have been made available in the current version of Bender for visualization of AIDA histograms: `rootPlotter` and `hippoPlotter` from LCG/PI project (visualization is performed using the ROOT and HIPPODRAW engines) and `OnXSvc` from PANORAMIX (using OPENSCIENTISTS [24, 25] engine). Also the visualization of histograms using the `PyROOT` extension module from native ROOT is possible. This approach exploits the fact that the internal transient representation of histograms in GAUDI uses the PI implementation of AIDA interfaces based on ROOT.

Visualization of event and detector data is possible through PANORAMIX. It is worth to mention that all these external packages act independently and can be freely intermixed within one BENDER session.

A single common interface for simple visualizations has been established:

```
>>> from benderXXXX import plotter
>>> plotter.plot( what )
```

where `XXXX` stands for `ROOT`, `PiRoot`, `PiHippo` or `Panoramix` and `what` could be any ROOT object (for ROOT), AIDA histogram (for all plotters), object or collection of objects from LHCb Event or Detector Model (for PANORAMIX).

The possibility to use GNUPLOT [26] and PYX [27] for visualization of histograms is currently under evaluation.

The interactive analysis of N-Tuples can be performed by using `PyROOT` if `ROOT` is chosen as N-Tuple persistency. The possibility of N-Tuple analyses within `GaudiPython` using HIPPODRAW has been demonstrated.

## SUMMARY

BENDER is a powerful interactive environment for development of physics analysis code. BENDER is part of the official LHCb software and is released regularly in both LHCb supported platforms `Linux` and `Windows`. The cur-

```
from bendermodule import *
class Dstar(Algo):
    def analyse( self ) :
        self.select ( tag = 'K-'  , cuts = ( 'K-'  == ID ) & ( PT > 1 * GeV ) )
        self.select ( tag = 'pi+' , cuts = ( 'pi+' == ID ) & ( P  > 3 * GeV ) )

        masscut = abs ( DMASS( 'D0' ) ) < 30 * MeV
        for D0 in self.loop ( formula = 'K- pi+' , pid = 'D0') :
            if ( VCHI2 ( D0 ) < 4 ) & masscut ( D0 ) : D0.save ( 'D0' )

        for Dst in self.loop ( formula = 'D0 pi+' )
                self.plot ( title = 'DM for D*+'                    ,
                            value = ( M ( Dst ) - M1 ( Dst ) ) / MeV ,
                            low   = 130 , hight = 180               )

        return SUCCESS
```

Figure 3: Same analysis as Figure 2 in PYTHON within the BENDER environment.

rent functionality provided in BENDER is sufficient for the needs of realistic physics analysis. Current application areas for BENDER include the physics analysis of Data Challenge 2004 Monte Carlo data and the studies for High Level Trigger.

## ACKNOWLEDGEMENTS

It is a great pleasure to thank S. Barsuk, J. van Hunen, S. Klous, P. Koppenburg, O. Leroy, G. Pakhlova, G. Raven, N. Root, H. Ruiz, J. van Tilburg, B. Viaud and M. Zupan for nice ideas,warm help and valuable feedback and H. Dijkstra, A. Golutvin, M. Merk and T. Ruf for the support, useful discussions and constructive criticism.

## REFERENCES

[1] S. Amato *et al.*, "LHCb Technical Proposal", CERN-LHCC-98-004

[2] G. Barrand *et al.*, "GAUDI: The Software Architecture and Framework for building LHCb Data processing Applications", Proceedings of CHEP'2000;

[3] M. Cattaneo *et al.*, "Status of the GAUDI event processing framework", Proceedings of CHEP'2001

[4] http://cern.ch/Gaudi

[5] I. Belyaev *et al.*, "Simulation Application for the LHCb Experiment", Proceedings of CHEP'2003

[6] G. Barrand, "PANORAMIX and LAJOCONDE. Interactive environments for LHCb", Proceedings of CHEP'2004

[7] http://cern.ch/lhcb-comp/Analysis

[8] http://www.python.org

[9] I.Belyaev, "LOKI: Smart & Friendly C++ Physics Analysis Toolkit", LCHb-2004-023

[10] http://aida.freehep.org

[11] http://cern.ch/LCG

[12] W. Lavrijsen *et al.*, "Reflection based PYTHON-C++ bindings", Proceedings of CHEP'2004

[13] http://cern.ch/root

[14] P. Kunz, "The HippoDraw application and the HippoPlot C++ Toolkit upon which it is built", Proceedings of CHEP'2001

[15] T. Glebe, "PATTERN - High Level Tools for Data Analysis", HERA-B-2002-002

[16] T. Glebe, "GCOMBINER 1.0", HERA-B-2001-001

[17] A. Alexandrescu, "Modern C++ Design: Generic Programming and Design Patterns Applied", Addison-Wesley, 2001, ISBN 0-201-70431-5

[18] http://cern.ch/wwwasd/lhc++/clhep

[19] http://www.boost.org

[20] http://cern.ch/seal

[21] http://cern.ch/pi

[22] C. Arnault, "CMT: a Software Configuration Management Tool", Proceedings of CHEP'2000

[23] http://www.cmtsite.org

[24] G. Barrand, "OPENSCIENTIST.Status of project", Proceedings of CHEP'2004

[25] http://lal.in2p3.fr/OpenScientist

[26] http://gnuplot-py.sourceforge.net
http://gnuplot.info

[27] http://pyx.sourceforge.net