Automatic Procedures as Generated Analysis Tool

Karen Abrahamyan, Galina Asova, Stefan Weisse, Michael Winde
Pedro Castro-Garcia, DESY, Hamburg and Zeuthen, Germany

ABSTRACT

The photo injector test facility at DESY Zeuthen (PITZ) was built to develop, operate and optimise photo injectors for future free electron lasers and linear colliders. In PITZ we use a DAQ system that stores data as a collection of ROOT files, forming our database for offline analysis. Consequently, the offline analysis will be performed by a ROOT application, written at least partly by the user (a physicist).

To help the user to develop safe filters and data visualisation (graphs, histograms) with minimal effort in an existing ROOT framework application, we provide a GUI that generates C++ source files, compiles and links them to the rest of the application. We call these C++ routines "Automatic Procedures" (AP).

Standard filter conditions and data visualisation can be generated by click or drag-and-drop, while more complex tasks may be expressed as small pieces of C++ code. Once compiled by ACLiC, an Automatic Procedure may be reused without repeated compilation. E. g. the injector shift crew will run a number of ROOT applications, controlled by APs in regular intervals. Alternatively every AP can be read in and loaded to the GUI for further improvement.

A number of APs can run in a logical sequence, parameters can be transferred from one AP to another. They can be selected by picking a point from a graph.

The GUI was constructed with Qt.

Keywords: ROOT, ACLiC, Automatic Procedure, Data Analysis, Data Visualisation, GUI, Qt

INTRODUCTION

PITZ is a Photo Injector Test Facility at DEZY-Zeuthen for research and development of laser driven electron sources for Free Electron Laser (FEL) and elector colliders.

Corresponding to the primary ideas of the facility, the PITZ Data Acquisition System (DAQ) is storing data in ROOT [1] files. These data represent the status of the facility at any moment and provides for offline analysis of single parts and correlations between different components as well.

In order to provide an easier and common way to access the data, we developed the DAQ browser *daqbr*.

The basic idea of this tool is to perform these tasks:

- to select data from the data-base
  - by the time this data was generated
  - according to filter conditions
- to derive new data (to perform calculations)
- to visualise results in the form of graphs and histograms
- to store the selected and derived data as a sequence of ROOT or text files for further usage by other applications

Additional tasks for *daqbr* are:

- to help physicists to safely develop sophisticated filters and routines for data visualisation on-the-fly
- to help physicists to create re-usable analysis code in a standardised way.

ARCHITECTURE AND DESIGN

Rich from the point of view of physics analysis tools (with its classes for storing, retrieving, processing and data visualisation) and being object orientated, which provides convenience in extensibility and maintenance, ROOT is used as a main tool in our offline analysis applications. Its framework orientation gives us the possibility to use parts of the existing code and to adjust them to our specific needs by inheritance.

On the other hand programming a complex GUI with ROOT is somewhat troublesome. Therefore we developed the major part of the Graphical User Interface with the tools provided by Qt [2].

Data analysis in PITZ is done mainly on SuSE Linux 8.2 machines, therefore *daqbr* was developed for this platform.

Principal Structure

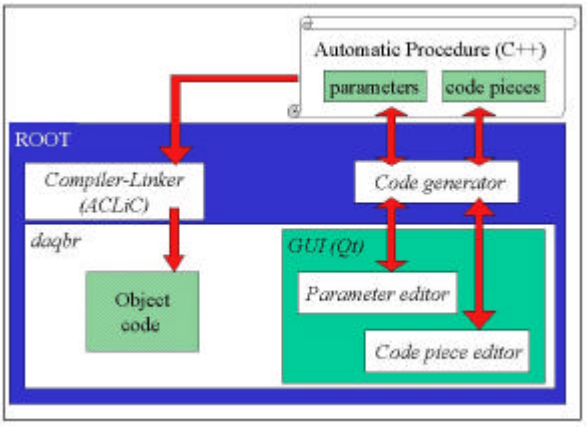

Figure 1: Principal structure of daqbr

As in any program, the user has to specify some parameters. To be able to code the calculations and decisions, *daqbr* in addition can manage user written snippets of C++ code. The parameters are specified via a graphical parameter editor, the code pieces via a text

oriented, context sensitive code piece editor. The parameters and the code pieces are sent to a code skeleton generator that creates an "Automatic Procedure" (AP) in the form of a C++ source file. This file is then compiled and linked by ROOT's compiler and linker ACLiC.

The Graphical User Interface

The GUI is separated into different tabs (Time, …, Filter, Plots and Histos, Output) according to the tasks one would like to define, although there is not a unique one-to-one correspondence between a tab and a task for *daqbr*. Filling the fields in the different tabs, the user specifies what should be done by the AP.

The first setting the user should make is to select the time period - when were the interesting data generated. It can be selected in two ways – absolute (defined beginning and end) or relative (one or both limits are set relatively to the time of running the application). The later one is useful e.g. in a shift's summary when we already have once created an AP and the only difference is the moment the physicists are interested in – no changes are necessary (Figure 2).

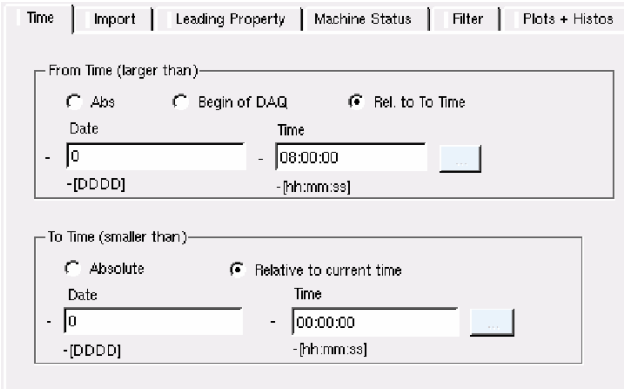

Figure 2: Selecting the last 8 hours before the AP runs

After the user has selected the time range, he will specify other filter conditions. Each event which falls in the time period will be tested whether it fulfils the conditions.

The names of all data stored are shown in an alphabetically ordered hierarchical list in the upper part of the GUI (Figure 3).

We distinguish standard (Figure 4) and C++ coded (Figure 5) filter conditions. An event passes the filter if it fulfils all standard conditions (need only values and names as parameters, are implicitly AND-ed) and C++ coded conditions (need calculations in addition). To perform these calculations one can use any C++ operations including OR (as shown in Figure 5). There are also some predefined functions, which can be achieved by clicking a button. A template of the respective function call together with its parameters will be inserted into the text. This way we tried to avoid that the user has to get special programming skills to manage the tool. It's enough for him to stick to two rules:

- get a value from the data base by specifying its DAQ name
- keep in mind to return false when filter tests fail

See Figure 4 and Figure 5.

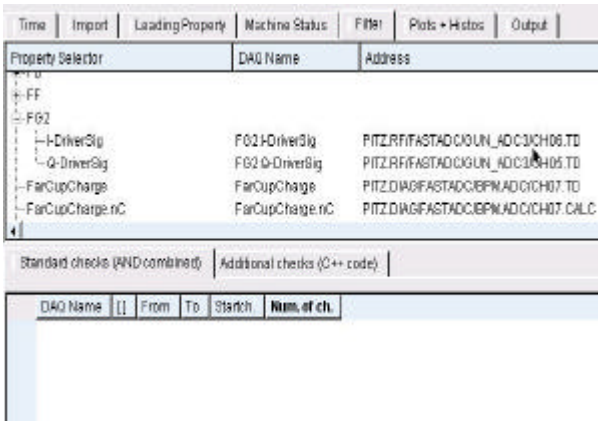

Figure 3: Structure of the GUI

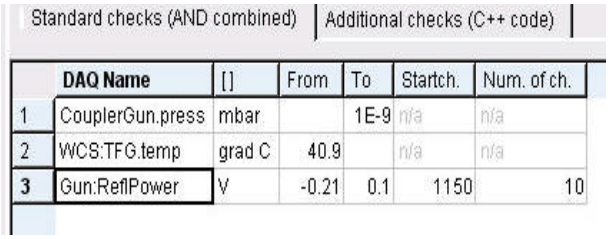

Figure 4: Standard checks

The DAQ name can be added to list of standard checks by drag-and-drop from the list of parameters in the upper part of the GUI. *From*, *To*, *Start Channel* and *Number of channels*, if they are necessary, must be inserted by hand. Drag-and-drop also works to the text field for the additional checks.

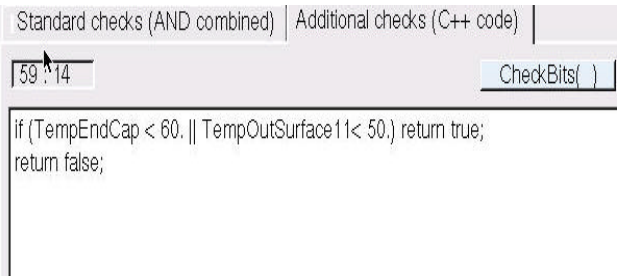

Figure 5: Additional checks – TempEndCap and TempOutSurface are names of variables stored in the data base, the test passes if at least one of them is in the range (OR)

The same techniques - drag-and-drop into tables and C++ text fields - are used in the other tabs to specify graphs, histograms and output.

The GUI provides saving and reloading of files. A reloaded file can be modified in the same way as it has been created.

DAQ variables versus DAQ properties

All the data in the data base are stored as so called properties - objects which contain information about their name, value, timestamp, size, etc. To hide these details from the user we introduced DAQ variables. A DAQ variable appears to the C++ code as an ordinary C++ variable. Suitable overloading of operators and functions provides for that.

DAQ variables also allow to smoothly manage "bit properties" (where an integer value is used for a number of status bits) – all the necessary masking and dealing with bitwise operators is done out of the user's awareness only once for an AP.

As a side effect we gain quicker execution of the program as well as a context sensitive code piece editor.

Why Automatic Procedure?

The GUI passes all parameters and user written snippets to a code skeleton generator which, according to a template, creates a C++ source – the AP itself. The code generator changes some names, inserts function calls, slightly modifies the snippets and inserts them at suitable places in the source. Once this file is created, it is compiled by ACLiC and linked to the rest of the application. Any warnings or errors are shown in the GUI itself. Code, complied and never changed, will only be linked when the same AP is being run for the next time. Here comes the term A*utomatic* from.

The code skeleton generator does not lead the user too down to the source level and keeps him aside from any complexities. It also serves as an auxiliary tool for the physicists, as it indirectly makes them create their programs in a common way and enforces some coding rules.

Structure of an AP

In each AP *My*.C the code generator creates two classes: T*My*Filter defining the filters and the data to be stored and T*My*PartialPlot defining the plots. These are derived from two classes – TAPFilter and TPartialPlot inside of the framework. TAPFilter and TPartialPlot are based on classes provided by ROOT – base and histogram classes, tree and container classes. It is worth to mention the usage of dCache [3] and ROOT's TDCacheFile which saves a lot of efforts and time for the user.

Every *My*.C, created by the code generator is still readable by the user, even if he is an occasional C++ programmer. All APs have the same structure and the programmer will easily find what he has written originally in the GUI.

The aim of filtering-storing is to get a file of manageable size for further processing. It is possible to extract meaningful data of an array. So the size of an array may vary in the output for different events for both ROOT and ASCII formats.

Plotting options concern history and correlation plots, 1DIM and 2DIM histograms simultaneously for standard and additional plots. All of them can be drawn on as many canvases as the user has defined. Special cases are interlock plots – a number of graphs is drawn on one pad.

A lot of tasks are error-prone, especially plot drawing and user coded parts. To reduce showing wrong results, *daqbr* widely uses C++ exceptions. The most important case and also hard to understand for the user, is that some of the data is not available for some time simply because it was not taken by the DAQ for that time. It is important to know why the problem has occurred during the data access but not everything depends on the user. His intervention is reduced to several actions depending on the severity of the error and its influence on the result. Error message like "File cannot be opened for reading" is the only case when he has to choose if he wants to get the full information from dCache, which is time consuming, or he would like to skip that period.

WHAT DID WE LEARN?

While the standard input options were quickly accepted by the users, the coded ones were still hard to manage for them. So we improved them in some steps, finally resulting in the context sensitive editor and the concept of DAQ variables.

There also were a lot of questions to be solved - how to generalise the original users' requirements so that a lot of tasks can be done; how to go through the data as fast as possible; what is the price in increasing the statistical output and so on.

ACKNOWLEDGEMENTS

We would like to thank Rasmus Ischebeck and Christian Lackas for the original implementation of the DAQ system at TTF, the PITZ team for the fruitful discussions and their support, the DOOCS and VUV-FEL teams. We would like to extend our thanks to the ROOT team for providing their excellent framework.

References:
[1] ROOT framework and RooTTalk Digest
http://www.root.cern.ch
[2] Qt references
http://www.trolltech.com/
[3] dCache
http://www-dcache.desy.de/