The Go4 Analysis Framework Introduction V3.3

J.Adamczewski, M.Al-Turany, D.Bertini, H.G.Essel, S.Linev
30 May 2007

Content

1 Editorial

Layout used in this document:

Text	Times New Roman, 10 pt
Verbatim text	`Courier new 10 pt`
Menu items	**Arial bold 9 pt**
Class names	*Arial italics , 9 pt*
Methods()	*Arial italics , 9 pt*

Go4 screenshots Style Window, Font Arial 11pt

Einfügen->Referenz->Querverweis: Überschrift+Überschriftnummer/Seitenzahl
Einfügen->Referenz->Index und Verzeichnisse: Eintrag festlegen, Indexeintrag+Aktuelle Seite. (search for Feld)
Index entries can be edited in text (first:second)
Index aktualisieren (RMB)
Inhaltsverzeichnis aktualisieren (RMB)

2 Release Notes

2.1 *New features in Go4 v3.03 (May07)*

1. **Viewpanel**
   a. **Marker editor:** A point- or region marker and its label will pop to the pad foreground when it is selected with left mouse button. Additionally, selection of a marker in the combo box of the editor will let it appear frontmost.
   b. In superimpose mode selected histogram can be moved on the top of complete histogram stack via new menu command "Select/show histo on top".
   c. **Draw options** enhanced: support for *TGraph* draw modes and *TGraphErrors* error style. Reorganization of draw options for TH1/TH2. New draw options tool for line, marker, and fill colours of histograms and graphs.
   d. **Menu "Select"** to chose active object from superimposed histograms and graphs.
   e. **Autoscale checkbox** as shortcut on top of each viewpanel
   f. **Improvement** in speed of view panel redraw (up to factor of 2).
2. **Fitpanel improvement:** keep y-scaling when fitting on x subrange of histogram
3. **New Zoom toolbar**: added buttons for scaling z-axis of 2d histograms.
4. **New icons for zoom toolbar** and draw options toolbar.
5. **New additional draw options** toolbar to select commonly used drawing options by buttons (lin/log, line, histo, some 2d styles). The new toolbar is displayed via the RMB options pull down menu.
6. **New example macro scalex.C** to scale x-axis of histogram with linear calibration function
7. **Settings menu**: "Show event status" selectable as default pad option.
   Settings menu: "Statistics Box..." dialog to define default pad options for histogram statistics.
8. **TGo4Picture**: new method *AddSpecialObject()* to add any ROOT graphical object (text labels, markers) to the picture
9. **Improvement** in *TGo4MbsFile* for partial read of lmd file: Corrected mismatch between first event index and real event number (before: index=event number-1).
10. **TGo4MbsFile**: now can also read list-mode data of old event formats type 4,1 and 4,2. Event will be converted implicitly into format 10,1 for further processing: User unpack processor gets *TGo4MbsEvent* with one *TGo4MbsSubevent* that contains all event data.
11. **GUI command interface** *TGo4AbstractInterface*. Added methods:
    - *GetViewPanelName()* - returns view panel name
    - *SetViewPanelName()* - changes view panel name
    - *RedrawPanel()* - updates view panel view
    - *RedrawItem()* - updates all views of specified items
    - *FindViewPanel()* - searches for view panel of specified name
    - *GetActiveViewPanel()* - returns currently active view panel
12. **Maintenance**:
    a. Some Makefile and build skript improvements
    b. Added missing includes for <math.h>, required by some compilers
    c. Due to changes in ROOT in many Go4 files includes like `TROOT.h`, `TMath.h`, `TList.h` are missing. Sometimes user should also include these files in user code.
    d. In latest ROOT *TBuffer* class becomes abstract, therefore one cannot use it directly in the code. Instead, *TBufferFile* class must be used.
    e. Adjustment of `Makefile` because of changes in default libraries for ROOT >= 5.13/04 (separated libSpectrum.so)
    f. Adjusted *Go4ThreadManager* package due to changes in *TTimer* copy constructor for ROOT versions > 5.12.00
    g. Some bug fixes concerning compilation against old ROOT versions 4.08
13. **Bug fix**
    a. for changes in ROOT>v5.14 pad cleanup: Viewpanel with go4 markers on subpads crashed when closed or cleared.
    b. 1-d histogram drawing. Due to some features of ROOT histogram painter several draw options (lin, barchart and others) not working after *TH1::SetSumw2()* is called - in there Sumw2 array sum of squares of weights is accumulated. Modification in Go4 code were done to avoid Sumw2 arrays when it not necessary.
    c. in Go4Socket library (missing include) because of changes in ROOT version 5.14-00
    d. Problems with view panel scaling functionality when build with gcc4.0.x compiler (FC5); fixed.

2.2 *New features in Go4 v3.02 (July06)*

1. **Analsis framework:** *TGo4EventElement* now implements default method *Fill()* that calls virtual function *TGo4EventSource::BuildEvent()*. As a consequence, for a simple analysis the user only has to implement *BuildEvent()* method in his processor class. There is no need to develop a user output event class. Even if a user output event class shall be used, methods *Fill()* and *Init()* are not necessarily needed for a standard analysis. `Go4ExampleSimple` and `Go4Example1Step` were changed accordingly.
2. **Analysis framework:** *TGo4EventProcessor* now implements *BuildEvent()* and can be used in steps which are only used as handle for event input (branched steps).
3. **Macro usage:** Analysis defines `__GO4ANAMACRO__` on startup to be used in any Go4 analysis script to check the current environment. In GUI, `__GO4MACRO__` is defined and can be checked analogously. In analysis, pointer **`go4`** is already set to *TGo4Analysis::Instance()*, in GUI to *TGo4AbstractInterface::Instance()*, i.e. all methods can be referenced by **`go4->`**. (see 6.4.3, page 49, and 6.17, page 79)
4. **Parameter editor** offers popup menu **GetFromFitPanel** for embedded fitters to update fitter settings from the current fit editor. Useful for calibration parameters that should be fitted interactively to spectra (see `Go4Example2Step`).
5. **Rebin in GUI**. Now when histogram will be rebinned via right-mouse menu or via ROOT graphical editor, rebinning will be kept when histogram will be updated next time from analysis. Many views of the same histogram with different binning are possible. Binning also kept in hot-start file. *TGo4Picture* has new *SetRebinX()*, *SetRebinY()* methods to configure rebinning of displayed histogram.
6. All **Go4 macros** put into new subfolder `$GO4SYS/macros`. This directory should be added to entry `Unix.*.Root.MacroPath` in `.rootrc` setup file.
   New macros: `savecond.C` and `saveparam.C` to create macros to set conditions and parameters to their current values (see 4.5.6, page 26).
7. **Bugfixes:**
   a. Access to RFIO root files from Go4 GUI browser was not possible (at GSI), since internal functions of `libRFIO.so` were shadowed by functions of GSI event lib with same names. Solved by separating Go4 event library package into different modules for analysis and GUI task.
   b. Analysis server executed *UserPostLoop()* each time a GUI client was disconnected. Disabled.
   c. Several changes concerning the cleanup mechanism in GUI object manager
   d. *AnalysisClient* in CINT mode showed thread deadlock for ROOT versions> 5.02-00
   e. Start client dialog selects correct analysis directory when choosing the analysis executable

2.3 *New features in Go4 v3.01 (May06)*

1. New **script command line widget for GUI**: Allows execution of ROOT commands or macros within Go4 GUI task. Moreover, Go4 hotstart scripts may be invoked here at any time. The widget offers a file dialog to search for *.C and *.hotstart files. It also has a selector dialog of preloaded commodity functions for histogram manipulation (rebinning, addition, projection, etc.). These function template calls may be completed with existing histogram names by dragging histogram items from the browser and dropping them on the empty command argument. The history of the command line may be saved to the current Go4 settings file `.go4/go4localrc` and is then restored on next startup. (See 6.17, page 79).
2. New **GUI command interface class** *TGo4AbstractInterface*. It can be accessed by handle "`go4->`" in GUI command line. This makes it possible to interact with Go4 GUI views and browser objects in a ROOT/Go4 script. Additionally, all remote analysis control commands are available here, like in the hot start scripts. Method reference of *TGo4AbstractInterface* is available in the Go4 help viewer (type "help" in GUI command line, or use **Help►GUI commandline** menu of Go4 main window). Example scripts using this interface are at `$GO4SYS/Go4GUI/scripts` (definitions of the preloaded command line histogram functions). **Note: have been moved to `$GO4SYS/macros` in V3.2.**
3. New **general marker label settings** dialog. In main window menu **Settings►Panel Defaults►Marker labels..**, a checkbox dialog offers to switch all label properties of the region and point markers (visibility and information displayed in the label). These settings have effect on all new markers of the view panel marker editor. They are saved in the go4 preferences file `.go4/go4localrc`. (see 6.7.6, page 62)
4. **Plain ROOT canvases** in files are better displayed.
5. New settings feature **Settings►Preferences►Fetch when saving**. If enabled, the **save browser / save memory** button of the file toolbar will refresh all browser item objects from analysis before saving. Thus the ROOT file will contain a snapshot of all analysis objects. Otherwise, only the already fetched objects are saved.
6. **Zoom tools** "set scale" dialog upgraded to non modal MDI widget. This will appear always on top of workspace widgets and refers to currently selected view panel pad. Changes include some bug fixes concerning the range settings of 2d histograms, and the auto-scale property.
7. **MBS monitor tool:** If monitoring switched on, calculation of rates is now done in Go4, averaged over update time. Parameters of MBS monitor are stored in Go4 settings file.
8. **TGo4Interface:** new method *ExecuteLine* to remotely do CINT call from Go4 master process in the remote slave process
9. **View panel superimpose mode** improvements:
    a. is not changed anymore after superimposed draw of FitPanel results, i.e. fitter data histogram can now be replaced just by drag and drop on the view panel
    b. existing axis labels of first histogram are kept
10. **FitPanel settings** are saved/restored in go4 settings file
11. **Fit GUI:** Enhanced draw styles for *TGraph*
12. **Bugfixes:**
    a. Workaround for ROOT crash in histogram rebin editor: Selecting a histogram in view panel for rebin with the ROOT attributes editor leads to segmentation violation when original histogram was replaced or deleted.
    b. **C**rash in Go4 markers/conditions when histogram in view panel was replaced by drag and drop.
    c. Update of histogram in GUI failed when histogram dimensions (ranges) were changed in analysis
    d. Position and size of histogram statistic label may now be saved in Go4 picture objects. Thus these properties can be restored on Go4 hot start.
    e. **C**rash on closing last non-minimized window in view panel
    f. **P**roblem with empty *TGraph* as data source in Fitter
    g. **C**rash when FitPanel histogram under work was replaced or deleted in view panel. FitPanel did not react automatically on changes, happening with histograms (or graphs), displayed on view panel. Therefore, when superimpose mode was switched off, fitted histogram disappeared from view panel (and also deleted), while fitter still has pointer on that histogram. Now FitPanel slot in object manager registered also against all histogram, used in fitting. If histogram is deleted, FitPanel will be automatically refreshed.
    h. Histogram title could not be switched off in superimpose mode in view panel
13. Improvements in make files
14. Adjustments of includes due to changes in new ROOT version 5.10

2.4 *New features in Go4 v3.00 (November05)*

1. Redesign of the GUI with **new internal object manager**. Decoupling of controlling functionality from the Qt graphics layer. Effects many of the following features.
2. **New Go4 browser**. Instead of several tabs for remote analysis, local memory, monitoring list, now one browser with sub-branches for different data sources, such as remote analysis, histogram servers, root files, is used. Supports local memory workspace folder with copy and paste by drag and drop, clipboard, and renaming. All controls available via right mouse button context menu. Switchable columns for object properties. Filter for monitored, fetched, and all objects.
3. **New view panel**. Improved marker editor with lightweight condition editor. Additional options to display date and time of refresh, and full object path. Can display same object with different draw styles and ranges simultaneously. May store current setup as Go4 picture.
4. **New condition editor**: More compact layout, shares functionality with view panel marker editor.
5. **Improved parameter editor**: May display user parameter structure without loading the user analysis library into the GUI. Suppresses display of unknown components.
6. **New dynamic list editor**: More compact layout. Automatic resolving of event name and data member name when dragging and dropping from analysis event structure, in case of pointer entry. Dito for tree name and draw expression in case of tree entry.
7. **New dockwindow for analysis terminal**. If analysis is started in external shell, functionality of analysis output window (macro execution, etc.) shrinks to dockwindow.
8. **Improved dialogs for analysis startup and connection**.
9. **Decoupling of libraries from GUI**. GUI does not require all analysis libraries anymore due to changes in command pattern and dependency rearrangements. Will speed up GUI startup time and may reduce memory consumption.
10. **Status monitor for remote MBS node**. New dockwindow offering connection to the mbs status port. Frequently update of daq rates and status possible. Trending histograms in browser workspace. Full printout of mbs status and setup structures possible.
11. **Go4 analysis status bar improved**. Animated Go4 logo shows true running state of analysis, independent of current event rate. Current event source of first active step displayed per name in text field.
12. **Remote control of Go4 analysis from regular ROOT session**. Command interface to connect and control analysis process from CINT. Inspecting and retrieving Go4 objects with extended root TBrowser possible.

2.5 *New features in Go4 v2.10 (June05)*

1. **Go4TaskHandler redesign**: Decouple client and server tasks from master and slave role. This implies that analysis can run in the network both as server or client task (as in previous Go4 versions). Vice versa, gui can run either as client or as server (previous behavior). Additionally, *TGo4AnalysisClient* class now inherits *TGo4Slave* (previously *TGo4ClientTask)*, and *TGo4Display* inherits *TGo4Master* (previously *TGo4ServerTask*). **One analysis server can be connected by many Go4 GUIs** (one controller/administrator GUI, and several observer GUIs).
2. **Go4TaskHandler redesign:** Password for login of master client to slave server with **accounts for administrator, controller, and observer** roles. Additionally, some Go4 commands are forbidden if master is logged in with a low priority account (observer e.g. may not reconfigure analysis, but only request objects for display). Default passwords may be changed in `MainUserAnalysis` code (see chapter 6.3.2 page 46).
3. **Go4GUI prepared to run with analysis server**: Command `go4 -client` will start the GUI master task in client mode. In this case, the **Launch analysis** dialogue requests for login account, password, node and connection port of the analysis server. Moreover, a client GUI may first launch a new analysis server in an xterm and connect to it afterwards (see chapter 6.3.2 page 46).
4. **Example of analysis server** in package Go4Example2Step: `MainUserAnalysis` may be started from command line with option `-server` as third argument (first arguments like batch, see 5.5.1,page 37), thus starting the analysis as server. Processing starts immediately (no submit from GUI necessary). Command line parameters of this example will set additional boolean arguments (servermode, autorun) of *TGo4AnalysisClient* constructor appropriately (see chapter 6.3.2 page 46).
5. **ROOT macro execution with Go4 analysis server:** A Go4 environment and analysis server can be started from any ROOT session in the background (`.x go4Init.C`). Go4 GUIs may connect to this server and request data from running analysis macros, or control macro via Start/Stop buttons. New methods *TGo4Analysis::WaitForStart()* to poll for the Go4 environment running state, and *TGo4Analysis::Process()* to invoke the Go4 analysis loop explicitely from ROOT macro (checks also for STOP). Example macros `hsimple.C`, `hsimplego4.C` and `treedrawgo4.C`. See chapter 7 page 80.
6. **Analysis:** *UserPreLoop()* and *UserPostLoop()* are only executed once when analysis running state is changing. In previous versions, each press on Start, or Stop button, respectively, would execute the corresponding method another time. Bugfix: postloop was called twice if analysis client was terminated in running state.
7. **Bugfix :** `MbsAPI/f_evt.c` (close of streamserver).
8. **Bugfix:** Labels for conditions and markers were not drawn correctly in logscale anymore for ROOT v>4.03/02.
9. **Bugfix:** Adjusted reallocation behaviour in *TGo4Socket* and *TGo4Buffer* to changed definition of *TBuffer::kIsOwner* flag for ROOT versions>4.03/02
10. Fixed several small memory leaks.

2.6 *New features in Go4 v2.9 (February05)*

1. **Keyboard shortcuts** for many functions (see table chapter 11, page 90).
2. **Settings for Go4 GUI** are now saved in the current directory by default in `$PWD/.go4/go4localrc` and `$PWD/.go4/go4toolsrc`, respectively. So different settings for the same login account are possible now. If the current directory does not contain a Go4 settings file on Go4 GUI startup, it will be created using the global account preferences at `$HOME/.qt`. Settings behavior can be changed using environment variable `GO4SETTINGS`. If this is set, the GUI preferences are used from directory `$GO4SETTINGS`. If `GO4SETTINGS` contains keyword `ACCOUNT`, the Go4 settings at `$HOME/.qt` are used (like in previous Go4 versions).
3. **New context sensitive menus** (right mouse button popup) for all GUI browsers.
4. **ROOT object editor *TGedEditor*** will show up in view panel side frame instead of top-level X-window. To implement this, the Go4 QtRoot interface has a new widget *TQRootWindow* which embeds a ROOT *TGCompositeFrame* into a *QWidget*.
5. **Superimposed drawn histograms**, *THStack* objects and *TMultiGraph* will show a ***TLegend* box** in view panel. The legend box can be switched on or off by view panel menu.
6. **View panel marker editor**: Added polygon shaped regions (*TCutG*).
7. **File browser: Added "Open remote file" functionality** to read objects from TNetFile/XRootd (ROOT:), TWebFile (http:), and tape library (rfio:).
8. **Analysis browser: Objects may be protected** against *Clear()* (histogram reset to 0), and against deletion in the analysis. Browser shows protection state in 3rd column as "C" and "D" symbols, respectively. Objects created from analysis code are always protected against deletion, objects created from GUI may be deleted from GUI again. Protection against clear may be changed using the browser's right mouse button menu. The protection state is persistent in the auto save file.
9. **Analysis:** Histograms associated with Go4 picture objects will not appear anymore in the analysis **Pictures** folder, but only in the **Histograms** folder.
10. **Analysis macro**: New analysis macro `MainUserAnalysisMacro.C` in directory `Go4ExampleSimple`. It needs a `.rootmap` file for automatically loading all necessary libraries. This file is created by the new files `Makefile` and `Module.mk` from the example. One can copy both files from the example, or modify existing files if they contain application specific changes. Look for `map-` expressions!
11. New Method ***TGo4Analysis::Print()*** to print the current setup of the analysis and the steps.
12. **Multiple input file (metafile) for *TGo4MbsFile* may contain lines with CINT commands** preceded by an "@" character. Commands, e.g. ROOT macro execution like "`.x setup.C`", are performed in between change of event source.
13. Metafiles should have suffix **`.lml`**. Then they are recognized without @. The main programs in the examples have been modified not to add a `.lmd` to a `.lml` file name (**update your main program accordingly**!).
14. ***TGo4FileSource*: Partial IO functionality** - name of the input event defines name of the tree branch to be read. Additionally, improved read performance for full event.
15. **New Example `Go4ExampleMesh`** to show how to setup an analysis with non-subsequent analysis steps. May use partial input from tree branch.
16. **Reorganisation** of Go4 make files and installation. Reduced number of Go4 libraries. Removed unnecessary ROOT dictionary information from libraries. Go4 may be installed without `libASImage.so` if this is not supported on the system.
17. **Implemented .rootmap mechanism** to auto-load required Go4 libraries in macros.
18. **Bugfix:** Preview panel options menu **apply to all** did not work for **histogram statistics** property.
19. **Bugfix:** Double click in Go4 GUI browsers was not always working, because of conflict with drag and drop mode.
20. **Bugfix:** When **Submit** was called without stopping the analysis before, references set in *UserPreLoop()* were not updated. Now *UserPreLoop()* is called also in this case. Additionally, *UserPostLoop()* is not called when analysis stops after initialization has failed.
21. **Bug fixes**: A set of use cases has been set up to test the GUI functionality. Several bugs have been found and fixed performing these use cases. The test procedure has improved the stability of the GUI. It will be extended and used for all future Go4 updates.

2.7 *New features in Go4 v2.8 (September04)*

1. **Marker editor** in view panel allows for marking channels or windows. Labels and arrows can be created. All marker elements can be saved and restored.
2. **New ROOT graphical editor** can be called from view panel. The editor dynamically adjusts to the graphical object selected by LMB.
3. **View panel window title:** can optionally be set by user and may be kept constant. If a *TGo4Picture* is displayed, the picture name defines the view panel title.
4. **Condition editor**: the cursor mode has been removed because the functionality is now provided by the markers
5. **Condition, markers and labels**: Implemented correct ROOT streamer (bug fix), i.e. saving and loading these objects to and from ROOT files is possible with fully recovered functionality and graphical properties. Support of pad display in linear and log scale (bug fix). Additional controls in RMB menu of ROOT (set ranges, location, save default properties, reset). Default label setup stored with Go4 GUI settings.
6. **Polygon condition**: Implemented statistics functions for work histogram under the cut (integral, mean, rms, etc.). Enabled **InsertPoint** and **RemovePoint** functions in RMB menu (bug fix).
7. **Fit GUI**: Selection between sigma and FWHM (default) by **Settings►Recalculate gauss width.** Fit results may be printed to terminal or Go4 log file output.
8. **1D drawing**: ROOT "L" (line) "C" (curve) "B" (bar chart) "P0" (poly-marker) line styles supported.
9. **Histograms**: re-binning, projections, and profiles supported (standard ROOT methods with RMB). Automatic "synchronize with memory" on pad click to get newly created histograms.
10. **Histogram client**: monitoring implemented (auto-update). Drag and drop support. Display error message when server connection is not available (bug fix). Store server specification in Go4 settings.
11. **File store**: Storing objects into a ROOT file a title is prompted. This title can be seen in the Go4 browser and the ROOT browser.
12. **UserObjects** folder: With *AddObject(...)* histograms, parameters and conditions can be put into folders of the **UserObjects** folder. They can be located there by the standard *Get* methods, e.g. *GetHistogram()*. Editors work also with objects in these folders. **Note: object names must be unique!**
13. **Log window:** Empty messages are now suppressed (bug fix).
14. **QtRoot interface:** bug fix concerning initialization order of X11 system (ROOT init now before Qt init). Lead to crash of the main GUI on newer Linux systems when using Qt versions > 3.1 (FEDORA2, SuSe9.1)
15. **Thread manager:** bug fix: adjusted default exception handling to work with newer `libpthread.so` that uses one process for all threads (e.g. FEDORA2). This lead to a crash when Go4 threads were canceled (shutdown of the go4 GUI).
16. **Analysis Framework:** bug fix: analysis without analysis step (*UserEventFunc()* only) again possible.
17. **Client startup script:** full `PATH` and `LD_LIBRARY_PATH` of the Go4 GUI environment is passed to the analysis process.

2.8 *New features in Go4 v2.7 (June04)*

1. **Keyboard shortcuts** (**Alt-1** to **Alt-5**) to select browser tabs (File, Monitor, Remote, Memory, Histogram client). Items are selectable with arrow keys (left-right to unfold and shrink subfolders). Return key acts as double click.
2. **MBS event classes improvements:** Method *TGo4MbsSubEvent::IsFilled()* checks if the sub-event was filled in the previous event built. Iterator *TGo4MbsEvent::NextSubEvent()* by default delivers newly filled sub-events only, suppressing existing sub-events in list of non used ids. Sub-event data field re-uses the memory allocated by `libgsievent` instead of copying it to own buffers. New method *TGo4MbsEvent::SetPrintEvent()* to set verbose mode for the next n events. Format changes in *TGo4MbsEvent::PrintEvent()*.
3. **Performance improvements** of analysis framework in step manager, dynamic list and MBS event classes.
4. **New EventInfo toolwindow** to control printout of an event sample in remote or local terminal. Optionally the user implemented *PrintEvent()* method, or the ROOT *TTree::Show()* output may be used. May control the arguments of *TGo4MbsEvent::SetPrintEvent()*. Supports drag and drop for event names from remote browser.
5. **Display total memory consumption** of histograms and conditions at the end of *PrintHistograms()* and *PrintConditions()* execution, respectively.
6. **TCanvas support in file browser improved:** Histograms saved inside a *TCanvas* in a ROOT file will appear in memory browser whenever this canvas is displayed
7. **Analysis Terminal window:** Limitation of text history buffer to 100 Kb by default, may be changed in settings menu. Disabled text wrapping in output for scrollbars.
8. **Scale values dialog window** extended by zmin and zmax fields. Allows setting minimum and maximum thresholds for channel contents of 2d histograms when auto scale is off.
9. Conservation of **TLateX textfields** when changing draw style or histogram statistics boxes visibility

10. File browser open file dialog allows **multiple file selection**
11. **Analysis configuration window:** remember path to previous selected file in event source, auto-save, and preferences dialogs. Some layout cleanups.
12. **Superimpose** of histograms with same name from different files possible if overwrite mode is deselected in memory browser. Histograms will be copied to memory browser with cycle numbers added to names.
13. **Bugfix:** Superimpose *THStack* does not crash anymore when deleting histograms
14. **Bugfix:** Crash after closing and re-opening view panel for same histogram with different sub-pad divisions
15. **Bugfix:** Analysis did stop when an analysis step without event processor is disabled
16. **Bugfix:** histogram bound to condition was not fetched from analysis when double clicking on remote condition icon
17. **Bugfix:** Double click on histogram in divided view panel did pop up this histogram magnified in a new view panel, but did not initialize view panel colours and crosshair settings correctly.

2.9 *New features in Go4 v2.6 (May04)*

1. **New Go4 Hotstart**: The current setup of the GUI (analysis name and settings, view panel geometry, objects in memory and monitor browser, displayed objects in pads) may be saved to a hot start script file (postfix "`.hotstart`") from the **Settings►Generate hotstart** menu. The script name may be passed as argument on next Go4 GUI startup (e.g. "`go4 mysetup`"), which will launch the analysis and restore the settings (e.g. from file "`mysetup.hotstart`").
2. **New *TGo4ExportManager*** class transforms and saves ROOT objects into other formats. Currently supported: plain ASCII (*.hdat, *.gdat) and Radware/gf3 (*.spe). An export filter is available in the GUI memory browser to save selected objects.
3. Redesign of **Go4 Auto-save** mechanism. Subfolders are mapped as *TDirectory* in *TFile* now, thus improving performance for large number of objects. Auto-save file is closed after each write, avoiding invalid file states in case of analysis crash. Dynamic list entries are saved as independent objects.
4. **Example macro `Go4Example2Step/convertfile.C`** converts all histograms and graphs from ROOT file into ASCII files, conserving the subfolder hierarchy.
5. **New *TGo4StepFactory*** class can be used as standard step factory to simplify the setup of analysis steps for small analyses. New example package Go4Example1Step shows the usage.
6. **The *TGo4Analysis*** class can now be used as standard analysis class. New example package `Go4ExampleSimple` shows the usage.
7. **New view panel** has size of previously active view panel. Default view panel starting size is stored in settings and recovered on next Go4 startup.
8. **View panel**: Switch on/off histogram title display in options menu.
9. **View panel**: Switch on/off crosshair for each pad in options menu. Default crosshair mode can be selected in main window settings menu and is saved and restored by Go4 settings. Crosshair mode button in condition editor has been removed.
10. **View panel**: Default background color can be selected in main window settings menu and is saved/restored by Go4 settings.
11. ***TCanvas*** objects in analysis task may be send and displayed on GUI. Works both for memory and monitoring list.
12. Support of ***TMultiGraph*** objects in analysis and GUI (display, memory and monitoring list update).
13. **New draw option *TASImage*** for 2 dim histograms in Go4GUI. May improve rendering speed for large maps when updating and resizing the canvas. Offers own palette editor in right mouse button popup menu.
14. **Parameter editor**: Added column to display the source code comments for each parameter class member as description.
15. **Condition editor**: General editor has button to create a new condition. New condition is defined in a dialog window and is put into general editor. May be sent to analysis for registration, or saved into a file then. All types of new conditions (window, polygon, array of these with variable size) are supported.
16. **Object editors** (condition, parameter, dynamic list) may save and load objects from/to ROOT files.
17. **Status messages** of object editors appear in bottom status line of Go4 main window.
18. Support of **dynamic list entries** in file browse: Editor opens on double click.
19. **Histogram and Condition info** windows: Object size now takes into account real data size on heap.
20. New analysis toolbar button for "**re-submit and start**" shortcut. Useful when file shall be re-read from the beginning after changing something in the setup.
21. **Auto-save** may be disabled completely from analysis configuration GUI.
22. New mode for ***TGo4MbsFile*** (*.lmd) wildcard/metafile input: Auto-save file may change its name whenever input file is changed. Name is automatically derived from input filename. Old behavior (one auto-save summing up all inputs) is still possible. This can be switched with method *TGo4Analysis::SetAutoSaveFileChange(bool)*.
23. **End of .lmd file** input gives informational message instead of error message.
24. **Bug fix: avoid log-file crash** when Go4 is started in directory without write access.
25. **Bug fix in Go4 Mainwindow exit** dialog. Exit via window "x" icon works properly now, too.

26. Some adjustments to work with ROOT versions > 4.00 in Go4Fit and qtroot packages

2.10 *New features in Go4 v2.5 (December03)*

1. Histograms may be bound to conditions by method *TGo4Conditions::SetHistogram()*. The bound histogram will be fetched automatically in GUI whenever condition is edited.
2. TGo4Picture can contain conditions together with histogram objects.
3. General condition editor in addition to the condition specific editors. Supports drag and drop of condition icons and conditions linked to TGo4Pictures.
4. Warning label for unsaved changes in condition editor, and in dynamic list editor.
5. Condition editor cursor tab can make copies of the current cursor marker. For printouts with multiple markers.
6. Analysis log window in GUI displays date and time of last refresh.
7. New histogram status window, and condition status window in GUI.
8. Redesign of GUI object management: Added drag and drop support of *TGraph*, TGo4Picture from all browsers. Bug fix and improvements in histogram superimpose mode.
9. Monitoring list supports *TGraph*, *TGo4Picture*, and *THStack*.
10. Logfile mechanism for GUI actions. Log output configurable in Settings menu. Logging output on demand from condition editor, histogram and condition status windows.
11. View pane can turn on or off histogram statistics box.
12. View panel supports fix/auto scale modes for *TH1*, *THStack*, and *TGraph* objects.
13. View panel resize speed improved (redraw only at the end of resize action). View panel does not start in full screen mode anymore.
14. Analysis terminal: New buttons for clearing the terminal, PrintHistograms, PrintConditions. Command line has shortcut "@" for "`TGo4Analysis::Instance()->`". "KillAnalysis" button buffered with confirmation dialog window.
15. "Quit Go4" button buffered with confirmation dialog window.
16. Dynamic list editor can change the global dynamic list interval for analysis.
17. Reorganization of GUI icons.
18. Performance improvements in TTimers of Go4 kernel: Removed Turn On/Off statements.
19. New method *TGo4Analysis::NextMatchingObject()* for search in analysis objects with wildcard expression.
20. Analysis: *PrintHistograms()*, *PrintConditions()* supports wildcard expressions for output list selection.
21. New methods: *TGo4Analysis::StoreParameter, StoreCondition, StoreFitter, StoreFolder* to write these objects into event store of an analysis step. Event number will be appended to object keys for parameter logging.
22. Consistency checks of analysis steps can be disabled by new method *TGo4Analysis::SetStepChecking(bool)*. For setting up of non serial type analysis steps with own user management.
23. *TGo4MbsEvent::PrintEvent()* extended to display headers and also data field contents of sub-events.
24. New methods: *TGo4MbsEvent::GetMbsBufferHeader(), TGo4MbsSource::GetBufferHeader()* to access the buffer headers of list-mode files. Implemented example in `Go4Example2Step`.
25. Go4 GSI histogram server also exports *TGraph* objects as histograms (if possible).
26. Implementation of *TGo4Condition::Paint()* to display Go4 conditions in regular ROOT environment. Conditions may be drawn on *TPad* which already contains a histogram. New classes for condition painters and condition views.
27. Reorganization of the distribution make files.

2.11 *New features in Go4 v2.4 (August03)*

1. New Package Go4Log to handle all messages and log file. This replaces the old package Go4Trace. Static method *TGo4Log::Message(char*, ...)* can be called everywhere to display text on terminal and optionally write to log file. Modified Go4 message prompt.
2. Header information of MBS list-mode data files accessible by new methods s_filhe* *TGo4MbsSource::GetInfoHeader()* and s_filhe* *TGo4MbsEvent::GetMbsSourceHeader().*
3. Event source class *TGo4MbsRandom* to deliver random spectra into MBS events without connection to MBS node or reading list-mode file. Matches event structure of standard example `Go4Example2Step`.
4. *TGo4Picture* objects can be used in the monitoring list.
5. Changes in Analysis configuration window: Number of events, start/stop/skip events may be specified; tag file name and optional socket timeout. File browser for event source files. Auto-save interval now refers to time (seconds) instead number of events. Modified layout.
6. Dynamic list editor with button to **PrintAll** dynamic list entries on analysis terminal.
7. Improved postscript print dialog in View-panel menu.
8. Histogram client API supports conversion into Radware format.
9. Go4 histogram server supports float histograms.

10. Execution of ROOT interpreter commands / macros in the analysis task possible by command line in analysis terminal window.
11. Re-design of condition editor:
    a. Display all conditions of array in different colors or hide them optionally. Visibility in editor is property of *TGo4Condition* and stored in auto-save file.
    b. Working view-panel pad and reference histogram of condition may be changed at any time.
    c. Clear counters button applies clearing to analysis condition immediately and refreshes editor from analysis.
    d. Statistics inside window condition limits (integral, maximum, mean, rms, etc) are calculated; these values are displayed in editor and may be drawn in labels on working pad. Methods to calculate statistical quantities belong to *TGo4WindowCondition* class and may be used in analysis, too.
    e. Cursor panel with crosshair mode and optional marker to pick values from displayed histogram. Cursor may be set by mouse click, by moving the graphical marker object, or by defining cursor position in the text fields. Cursor values may be drawn in label on working pad
    f. Extension of polygon condition /*TCutG* is calculated and shown like the borders of the window condition.
    g. Improved creation of new *TCutG* functionality. Assignment to current polygon condition may be cancelled. Handles pads with multiple *TCutGs*.
12. Added class *TXXCalibPar* to `Go4Example2Step`. Shows a procedure how to calibrate spectra using the Go4 fitter in connection with the parameter mechanism and an ASCII file "database" of line energies.
13. Make full screen default for new view panels.
14. When updating objects in Memory folder, a redraw is done automatically.
15. When monitor updates a View-panel, the pads are updated without blocking the GUI (not yet for picture)
16. Button besides zoom buttons to enter display limits by values
17. Drag pictures from Analysis pad to View-panel (only empty view panel, or is inserted in pad)
18. Some buttons on the browser pads have been rearranged to be consistent. On Memory browser pad the icons for "update local objects" and "synchronize with directory" have been exchanged to be consistent with Analysis pad.

2.12 *New features in Go4 v2.3 (May03)*

1. *TGraph* objects can be registered and displayed correctly. Reset of *TGraph* (clear all points) by "eraser" button from GUI possible.
2. Reset/clear complete folders by selecting them in remote browser and "eraser" button. New method *ClearObjects("Histograms")* to reset all objects of named folder, e.g. all histograms at once.
3. "Print" button to printout histogram and condition lists with statistics in analysis terminal. These buttons are located in the dynamic list editor.
4. Parameter classes may contain *TGo4Fitter** references or arrays of these. Fit GUI can be used to edit fitter from within parameter editor. Framework provides new class *TGo4FitterEnvelope* as example parameter. Example put into *TXXXAnalysis*.
5. User defined event source is possible. New class *TGo4UserSourceParameter* to be checked in analysis step factory for any kind of input. Example package `Go4ExampleUserSource` shows usage.
6. New class *TGo4Picture* to define layout of canvas with histograms. Pictures are registered in Go4 Pictures folder and stored in auto-save file like histograms; they can be displayed in any view-panel. Example added in *TXXXAnalysis*.
7. Possibility to register complete *TCanvas* objects in Go4 Canvases folder to be saved within auto-save file. Switch *TGo4Analysis* into ROOT batch mode to suppress drawing actions in analysis client while canvas is set up.
8. Go4 GUI can display and compare objects from different files in the same view panel now.

2.13 *New features in Go4 v2.2 (April03)*

1. Possibility to select rsh or ssh and analysis output in Xterm or GUI window.
2. Wildcard in input lmd file names.
3. Input file name beginning with @ is interpreted as text file containing lmd file names.
4. An auto-save file can be written on demand (button in configuration menu).
5. Parameter editor. User parameter objects (subclasses of *TGo4Parameter*) registered in the analysis can be edited in the GUI by double click in the browser. Currently supported members are the primary data types and arrays of these.
6. New environment variable `GO4USERLIBRARY` can be set to a colon separated list of ROOT user libraries which are loaded automatically in the GUI. This is needed for editing parameter objects.
7. Dynamic lists. A dynamic list editor can be used to create/specify dynamic entries. A dynamic entry consists of a histogram (can be created new) and a member of an event object which shall be histogrammed. Optionally a condition can be added. The condition also can be created new. The event structure is expanded in the browser. Drag&drop is provided to select members.

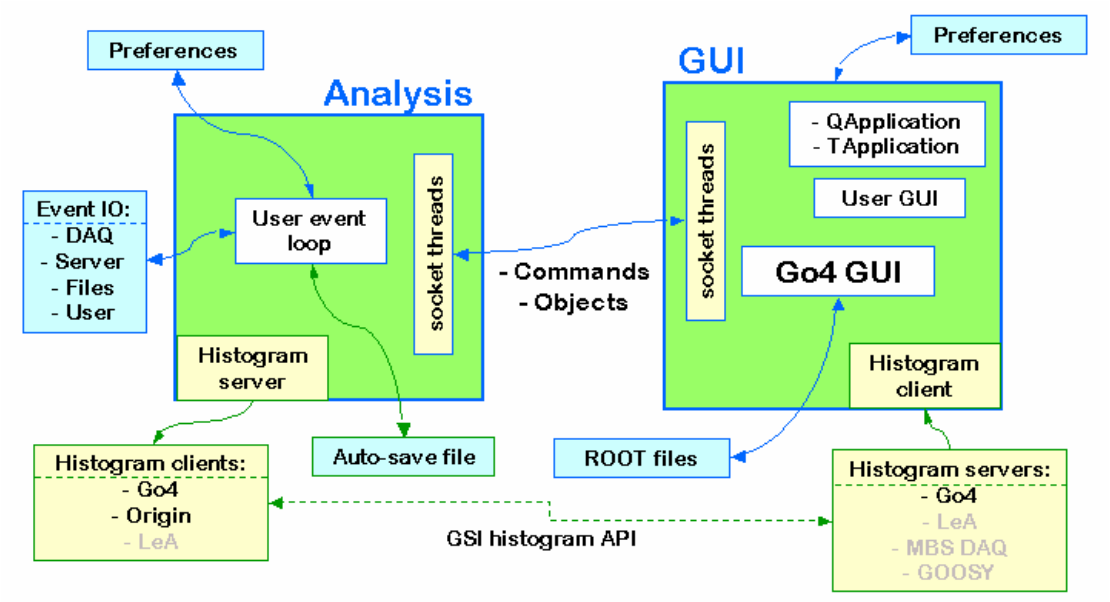
8. The condition editor has been improved. Arrays are now handled properly. *TCutGs* for polygon conditions can be created new.
9. *TGraph* objects are supported like histograms.
10. In the Go4 view panel, the ROOT "event status" (cursor position) can be displayed.
11. The new fit GUI is available. It includes three different peak finders, a simple fitter, a wizard, and full access to all fitter components. Fitters can be stored/retrieved to/from files or memory.
12. User Makefile: the user executable need to be linked against the make file variable `$(GO4LIBS)` only, as defined in the `Makefile.config` of the framework (see `Makefile` of example `Go4Example2Step`).

3 Introduction

The Go4 (GSI Object Oriented On-line-Offline) Analysis Framework has been developed at GSI. It is based on the ROOT system of CERN. Therefore all functionality of ROOT can be used.

3.1.1 Go4 tasks with all communications

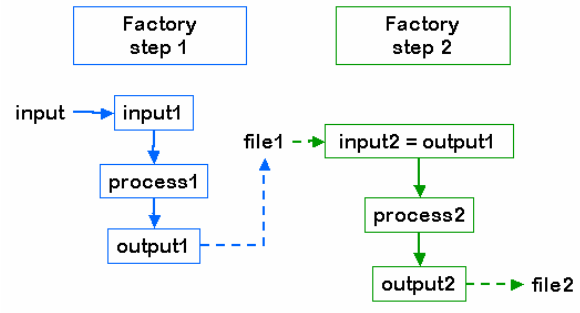
Go4 has two parts: the analysis framework itself and a Qt based GUI. Both can be used independently, or together. The separation of the analysis and GUI in two tasks is especially useful for on-line monitoring. The analysis runs asynchronously to the GUI which is (almost) never blocked. The same analysis can be run in batch/interactive mode or in remote GUI controlled mode. The GUI can be used stand alone as ROOT file browser and as histogram viewer for GSI standard histogram servers like MBS. Moreover, the analysis task can be run either as a client bound to one GUI (default), or can be started as an analysis server with the possibility to connect several GUIs (one controller and arbitrary number of observers with restricted commands).



gui150

3.1.2 Go4 analysis steps

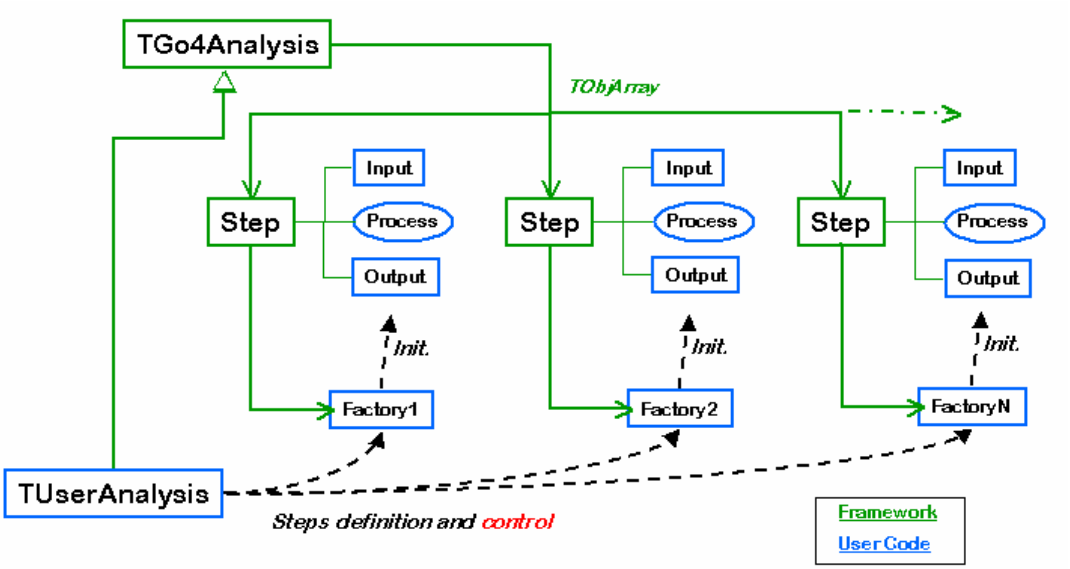
The Go4 framework handles event structures, event processing, and event IO. The analysis event loop is organized in **steps**: Each step has an **input** event, an **output** event, and an event **processor**. The output event calls the event processor to be filled. The event processor has also access to the input event. In the current design the analysis is data driven. A first event object (input1) is filled from some event source (input). An output event object (output1) is filled by an event processor object (process1) which has access to both, input1 and output1. Optionally the output event may be written to a file (file1). In the next step the input event object (input2) can be either the output event object (output1) from the previous step or retrieved from the file. The second output event object (output2) is filled by the second event processor object (process2) and can be optionally written to a second file.



gui147

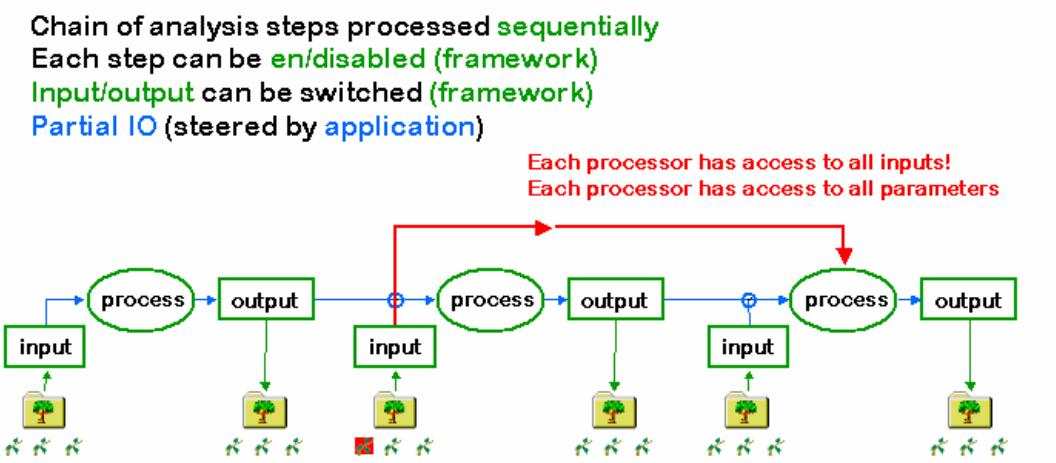
The information needed to create the event and processor objects (which are deleted when the event loop terminates) is stored in step **factories** which are kept in the **analysis**.

The processor and output event classes have to be provided by the user. The input classes for standard GSI event sources are provided by Go4 (see chapter 4, page 21). Analysis and step factory classes are provided by Go4 or can be implemented by the user as subclasses.



gui148

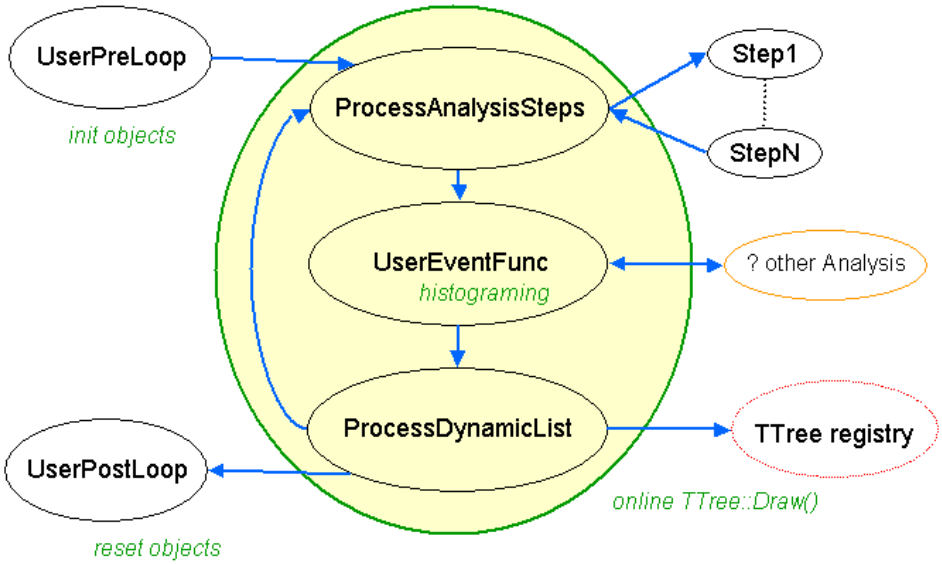
For normal operation, the Go4 analysis steps are designed to run subsequently. But in addition, each analysis step has access to the output events of all other previous analysis steps, so it would be possible to let analysis steps logically run "in parallel", all starting with the output event of the first step, and all delivering their results to the last step that may collect and combine them.



gui149

3.1.3 Other analysis functions

Outside the analysis steps the user functions *UserPreLoop(), UserPostLoop(),* and *UserEventFunc()* located in the user analysis class are executed as shown in the figure. In principle, they could be used to implement the full analysis without using the step mechanism. But for setting up a new analysis the use of steps is strongly recommended, because steps can be controlled by the GUI and offer event and IO management.
In the event loop, after processing the steps and *UserEventFunc()* the Go4 dynamic list processor is executed. This processor can be dynamically configured from the GUI to check conditions and/or fill histograms.



gui146

4 Go4 Analysis

The Go4 concept consists of base classes (interfaces) for event structures, algorithms, and IO, which can be implemented by user subclasses or by framework plug-ins (general service classes) delivered with Go4. Class descriptions and reference guides are available on the Go4 Website **http://go4.gsi.de**.

4.1 *Event base classes*

The interface classes provided by Go4 (a detailed description is in the reference manual) are normally not seen by the user. Starting with the examples (see chapter 5, page 30) one can better study derived working classes.

TGo4EventElement: Defines the event structure and methods to clear and fill this structure. Input and output event structures of each step of the analysis are instantiated once at initialization. In the event loop the virtual methods *Fill()* and *Clear()* are used to update the event data. These methods can be implemented in the user subclass. By default, *Fill* calls *BuildEvent* of event processor.

TGo4EventSource: The source of the event data. This can be e.g. a file of a certain format, or a socket connection to an event server. Usually, the event source class has a *BuildEvent(TGo4EventElement*)* method, e.g., which can be called by the *Fill()* method of the event object to be filled with the data. Therefore, event element and event source implementation classes have to "know" each other to perform a matching fill procedure. The class constructor should open (connect) the source; the destructor should close (disconnect) it properly.

TGo4EventStore: An object responsible for storing the event data. This can be e.g. a local file of a certain format, but may as well be a connection to some storage device. The virtual method *Store(TGo4EventElement*)* is used to store the pointed event object. The class constructor should open the storage; the destructor should close it properly.

TGo4EventProcessor: An object that contains the algorithm to convert an input event object into an output event object (both of class *TGo4EventElement*). This is a subclass of *TGo4EventSource*, since it delivers the filling of the output event from the input event. The event processor implementation has to "know" the input and output event classes. The methods of converting the data (i.e. actually performing the analysis) are free to be defined by the user. By default a *BuildEvent* method is provided.

TGo4EventFactory: Defines the actual implementations of all the above. Go4 uses a factory design pattern to create all event class objects at initialization. The virtual methods:
CreateInputEvent(), CreateOutputEvent(), CreateEventSource(TGo4EventSourceParameter), CreateEventStore(TGo4EventStoreParameter*), CreateEventProcessor(TGo4EventProcessorParameter*)* have to be defined in the user factory. They create the respective objects and return the pointer to it. The default factory provides methods *DefEventProcessor(objectname, classname), DefInputEvent(objectname, classname)* and *DefOutputEvent(objectname,classname)*.

Simple examples of a running Go4 analysis can be found on directories `$GO4SYS/Go4ExampleSimple`, `$GO4SYS/Go4Example1Step`, and `$GO4SYS/Go4Example2Step`.

4.2 *Event classes, interface to MBS*

Go4 offers predefined implementations of the event base classes, including an interface to the GSI data acquisition Multi Branch System MBS, the GSI list-mode files, and ROOT files.

TGo4EventElement (base class):

TGo4MbsEvent,	
TGo4MbsSubEvent	MBS event format 10-1
TGo4CompositeEvent	Base class for all composite event structures
TGo4ClonesElement	Clonesarray container for composite event

TGo4EventSource (base class):

TGo4MbsFile	(read from *.lmd list-mode file with format 10,1)
TGo4MbsEventServer	(connect to MBS event server)
TGo4MbsStream	(connect to MBS stream server)
TGo4MbsTransport	(connect to MBS transport server)
TGo4RevServ	(connect to remote event server)
TGo4FileSource	(read from *.root file from Go4 tree, i.e. one file containing one *TTree* per analysis step)

TGo4EventStore (base class):

TGo4FileStore	(write to *.root file with Go4 tree, this file can be used as *TGo4FileSource* later)
TGo4BackStore	Use *TTree* existing only in memory to view and analyze event structures.

These classes can be used directly to write simple analysis.

4.2.1 A simple event loop

Using these implementations, getting MBS event data into ROOT (without Go4 framework) could look like this:

```
#include "Go4EventServer/Go4EventServer.h"
#include "Go4Event/TGo4EventEndException.h"
int main() {
      TGo4EventSource* input = new TGo4MbsFile("file.lmd");      // MBS list-mode file
   // TGo4EventSource* input= new TGo4MbsTransport("node");   // MBS transport server
   // TGo4EventSource* input= new TGo4MbsStream("node");      // MBS stream server
   // TGo4EventSource* input= new TGo4MbsEventServer("node"); // MBS event server
   // TGo4EventSource* input= new TGo4RevServ("node");        // Remote event server
   TGo4EventStore* output = new TGo4FileStore("output",1,5);  // split level, compression
   TGo4MbsEvent* event = new TGo4MbsEvent();
   event->SetEventSource(input);
   event->Init();
   Int_t eof = 0, numEvents = 0;
   while(eof==0) {
      try{
         event->Fill();                                       // read event
         numEvents++;                                         // eof throws exception
         output->Store(event);                                // write to file
      }
      catch(TGo4EventEndException& ex) { eof=1; }             // mark end of file
      catch(...) { cout << "Error" << endl; eof=2; }          // any other error
   }
   cout << "EOF after " << numEvents << " events" << endl;
}
```

The events in the ROOT file can be retrieved by program, but not in tree viewers. For the use of tree viewers, a new output event object should be filled and stored.

4.3 *Analysis step classes*

As mentioned above a Go4 analysis is organized in steps. The information needed to instantiate a step is kept in the step factory.

TGo4EventFactory (base class):

TGo4EventServerFactory (base class): (contains factory methods that already know the above implementations. **User step factories must inherit from this class**!)

TGo4StepFactory This *TGo4EventServerFactory* can be used in most cases as user factory to set up the analysis steps (examples `Simple` and `1Step`).

TGo4AnalysisStep objects of this class hold the definition of an analysis step.

Each analysis step has at least an input event object, an output event object and an event processor object. Additionally, it can have an event source (e.g. *TGo4FileSource*) and an event store (*TGo4FileStore*) instance. An analysis step is set up by a *TGo4EventServerFactory* subclass, i.e. *TGo4StepFactory* or a user defined subclass.

4.4 *Analysis base class*

Once the user has defined his/her event class implementations, the analysis steps can be created and registered to the Go4 analysis framework. The actual framework consists of the ***TGo4Analysis*** class, which is a singleton (i.e. there is only one framework object in each process). This class provides all methods the user needs, it keeps and organizes the objects (histograms,...), it initializes and saves the data objects.

The user analysis is set up in a subclass of *TGo4Analysis*, i.e. ***TUserAnalysis***. Constructor and destructor of this user class, in addition with the overridden virtual methods *UserEventFunc(), UserPreLoop(),* and *UserPostLoop()* specify the user analysis.

If the latter functions are not needed, one can also use the *TGo4Analysis* class directly, as shown in the example `Simple`.

In the constructor of the *TUserAnalysis* class the analysis step objects are created, each containing instances of its user step factory. The analysis steps are registered at the *TGo4Analysis* framework, input and output events of subsequent steps are checked for matching. Furthermore, other objects like histograms, conditions or parameters can be created in the constructor and registered, so the framework is responsible for their persistence. Such objects can also be created in the step processor.

In addition to the event processors, the *UserEventFunc()* allows the user to specify analysis operations that are called once in each analysis cycle, e.g. filling certain histograms from the output events of all analysis steps. The *UserEventFunc()* makes it even possible to call an external analysis framework event by event without using the Go4 Analysis Steps at all, thus taking advantage of the Go4 object management and remote GUI features.

The *UserPreLoop()* and *UserPostLoop()* functions may define actions that are executed before starting, or after stopping the main analysis loop, respectively.
Once the user analysis class is defined, there are two modes of operation: The single-threaded batch mode, and the multi-threaded client mode that connects to the Go4 GUI.

4.4.1 TUserAnalysis example

The constructor of a TGo4Analysis derived user class could create one analysis step with input from an MBS file with the following code fragments (note that we use the standard Go4 step factory class):

```
TUserAnalysis::TUserAnalysis() {
//...
TGo4MbsFileParameter*      input;
TGo4StepFactory*           factory; // standard factory provided by Go4
TGo4AnalysisStep*          step;

input         = new TGo4MbsFileParameter("file.lmd");
factory       = new TGo4StepFactory("Factory");
step          = new TGo4AnalysisStep("Analysis",factory,input,0,0);
// the objects specified here will be created by the framework later:
factory->DefEventProcessor("XXXProc","TXXXProc");// object name, class name
factory->DefOutputEvent("XXXEvent","TXXXEvent"); // object name, class name
step->SetSourceEnabled(kTRUE);
step->SetProcessEnabled(kTRUE);
AddAnalysisStep(step);
//...
}
// Example of using the event loop functions for a trivial counting of events
// fEvents must be defined in TUserAnalysis.h:

Int_t TUserAnalysis::UserPreLoop() {
       fEvents=0;
       return 0;
}
Int_t TUserAnalysis::UserEventFunc() {
       fEvents++;
       return 0;
}
Int_t TUserAnalysis::UserPostLoop() {
       cout << " Total events: " << fEvents << endl;
       return 0;
}
```

4.5 *Main analysis program*

Typically the user provides the main analysis program. One can use one of the examples. The main program sets up the analysis. Then it starts two different modes (see example below):

4.5.1 Batch or command line mode

In batch mode, the constructor of the user analysis class (e.g. ***TUserAnalysis*** or ***TGo4Analysis***) creates the framework.

The *InitEventClasses()* method uses the factories of all steps to create the event classes and open the event sources, event stores, etc. It also checks for consistency of subsequent steps.
The *RunImplicitLoop(Int_t n)* calls the implicit event loop and runs the main analysis cycle (event processing of all enabled steps, *UserEventFunc())* for n times.
CloseAnalysis() deletes all event classes and closes all input/output files and connection. This method is complementary to *InitEventClasses()* that creates them.
The destructor of the user analysis class calls *CloseAnalysis()*; in addition the auto-save file is closed and the complete framework is shut down.

4.5.2 Client mode controlled by Go4 GUI

In the interactive GUI mode, the analysis framework is created with the user analysis class object, as for the batch mode. Additionally, the framework is handed over to a ***TGo4AnalysisClient*** object that manages the connection to the GUI. Usually, the Go4 GUI is started first and launches the analysis framework in a remote shell. The user analysis program is called in the shell script `AnalysisStart.sh` in the user's working directory. The working directory as well as the name of the executable is passed from the GUI side. Then the user executable creates the analysis framework and connects the multi-threaded analysis client to the Go4 GUI. After the connection is established, the complete analysis framework can be controlled from the GUI. After the example, we describe in detail what is happening on startup of the analysis client and what effect the GUI control actions have.

4.5.3 Analysis in server mode for multiple Go4 GUIs

As a default, the ***TGo4AnalyisClient*** object will set up the analysis-GUI connection in a way that the analysis is a single client to a single GUI as server, as described in section 4.5.2. However, it is possible to run the analysis as a server to connect many GUIs (one controlling GUI and many observer GUIs). Still the analysis class object is handed over to the ***TGo4AnalysisClient*** object, but the analysis "client" may run in a network server mode by constructor parameter (note: the classname ***TGo4AnalyisisClient*** was not changed for backward compatibility , although it should rather be called ***TGo4AnalysisSlave*** to point out the role as command receiving entitiy).
The analysis server may be started independently from the GUI from a shell like in the batch mode, and may already start analysis run from preferences setup without any controlling GUI. However, it can as well be launched from the *StartAnalysis* dialogue of the Go4 GUI when selecting the mode "as server". A Go4 GUI is ready to connect any such started server. Login of GUI to the analysis server may be with observer, controller, or administrator role, respectively; their passwords can be set in user analysis code. There can be only one controller or administrator, but multiple observer GUIs. Observers may only view exisiting objects, but may not modify them or change analysis setup and running state. Controller may view and modify objects and analysis configuration, but is not allowed to terminate analysis server. Only Administrator may shutdown the analysis server, too.
Additionally, the analysis server may be launched first from one GUI in an xterm, and then connected from this GUI and other GUIs later on. See section 6.3.2 for more details on connection of the GUI client.

4.5.4 MainUserAnalysis example

The following examples show the essential structures to start/run the analysis. See also the running examples.

Using user event and processor classes, but others all Go4 standard:

```
TROOT go4application("GO4","Go4 user analysis");  // initialize ROOT

int main(int argc, char **argv) {
  TApplication theApp("Go4App", 0, 0); // ROOT application loop
  // ...
  TGo4Analysis*     analysis = new TGo4Analysis::Instance();
  TGo4StepFactory*  factory  = new TGo4StepFactory("Factory");
  TGo4AnalysisStep* step     = new TGo4AnalysisStep("Analysis",factory,0,0,0);
  analysis->AddAnalysisStep(step);
  step->SetEventSource(new TGo4MbsFileParameter("myfile.lmd"));

// tell the factory the names of processor and output event
// both will be created by the framework later
// Input event is by default an MBS event
  factory->DefEventProcessor("XXXProc","TXXXProc");// object name, class name
  factory->DefOutputEvent("XXXEvent","TXXXEvent"); // object name, class name

  if (strcmp(argv[1],"-gui") == 0) // was started from GUI: create analysis client
  TGo4AnalysisClient* client = new TGo4AnalysisClient(argc,argv,analysis,kFALSE,"","");
  else { // run implicit event loop
     analysis->InitEventClasses();
     analysis->RunImplicitLoop(100000); // number of events
     delete analysis;
     gApplication->Terminate(); // exit
  }
  theApp.Run(); // needed to run the client, or if gApplication was not terminated
} // end main
```

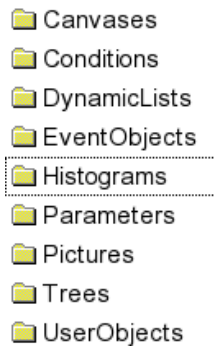
Using user analysis, event and processor classes (steps set up in *TUserAnalysis*):

```
TROOT go4application("GO4","Go4 user analysis");  // initialize ROOT

int main(int argc, char **argv) {
  TApplication theApp("Go4App", 0, 0); // ROOT application loop
  // prepare arguments for TUserAnalysis
  // ...
  TUserAnalysis* analysis = new TUserAnalysis(...); // arguments as required

  if (strcmp(argv[1],"-gui") == 0) // was started from GUI: create analysis client
  TGo4AnalysisClient* client = new TGo4AnalysisClient(argc,argv,analysis,kFALSE,"","");
  else { // run implicit event loop
     analysis->InitEventClasses();
     analysis->RunImplicitLoop(100000); // number of events
     delete analysis;
     gApplication->Terminate(); // exit
  }
  theApp.Run(); // needed to run the client, or if gApplication was not terminated
} // end main
```

4.5.5 Go4 objects



gui151

Objects used in Go4 are organized in ROOT folders. The folder structure is sent to the GUI. Objects must be registered in the analysis to be seen in the GUI browser. Registered objects can be located in the processors. The top folders as seen in the GUI are shown on the left side. The methods to register/locate objects are (pointer to the appropriate object, optional subfolder as string, name including subfolder as string):

- *AddHistogram(pointer,subfolder), GetHistogram(name)*
- *AddAnalysisCondition(pointer,subfolder), GetAnalysisCondition(name)*
- *AddParameter(pointer,subfolder), GetParameter(name)*
- *AddPicture(pointer,subfolder), GetPicture(name)*

These methods are available in ***TGo4Analysis*** and ***TGo4EventProcessor*** subclasses. Objects created in a *TGo4Analysis* subclass can be located in all event processors. Objects created in event processors can be located in all subsequent event processors (steps).
Registered objects are stored/retrieved to/from the auto-save file, if enabled. Retrieval is done **after** creation of the analysis singleton **before** the creation of the steps. When an object retrieved from the auto-save file is created in a processor the retrieved object is replaced (stored data lost). When an object is created in the analysis singleton it will be replaced by the one retrieved from the auto-save file except histograms which are not retrieved in this case. This means that histograms created in the analysis singleton are always empty after startup.

4.5.6 Go4 parameters

Parameters used in the analysis are implemented by the user in classes derived from ***TGo4Parameter***. Such objects are registered to the framework and can be edited by a generic parameter editor (see chapter 6.11.2, page 72). Parameter objects can be created in the user analysis or the event processor class. Parameter objects are loaded from an optional auto-save file after instantiation of the analysis and before instantiation of the processor objects. When created in the analysis the values set in the constructor are therefore overwritten by auto-save. To use the GUI editor, the *UpdateFrom()* method must be implemented to update the local (active) parameter object from the modified one delivered by the editor. In this method it is up to the user to ignore certain members or to execute whatever he wants. E.g. one could use parameters to execute commands. Parameters in the auto-save file can be edited. In the editor they can be saved/retrieved to/from files. Several mechanisms can be implemented to handle the parameter member values. The main question is how restricted the methods of modification should be.

1. Modify values only in the class constructor, then recompile. To prohibit changes by editor, the *UpdateFrom()* method could be just a no-op to avoid undocumented changes. The parameter object should be created and registered in the processor constructor (after possible auto-save restore). Pro: the parameter values are always strictly defined as coded. Con: the parameter values cannot be changed easily.
2. Modify values by editor, use auto-save to store. Create parameter object in analysis constructor. Auto-save must be enabled. Pro: parameter can modified by editor (*UpdateFrom()* method must be implemented) and changes will be restored from auto-save. Con: when the auto-save file must be deleted for some reasons. the latest values are lost.
3. Use a macro to set values. This macro must be executed in the processor constructor (after auto-save restore). *UpdateFrom()* could just execute the macro to avoid undocumented changes. Pro: values are kept in a text file and can be modified without recompile. Con: parameter cannot be changed by GUI editor.
4. Best combination: one can use macro `saveparam.C([file],wildcard,prefix)` from `$GO4SYS/macros` creating macros (one per parameter) to set all parameters to their current values,. The names are built from prefix and parameter name. The macro can be executed in CINT (then the parameters are taken from a file), or in the GUI or in the analysis. The parameter is created in the analysis. Values are set from macro in processor constructor. By this method parameter values can be edited by GUI, or macro can be edited. Last version will be used independently of auto-save.
   Example:
   `root[0].x saveparam.C("myfile.root","*","setpar")`
   would produce macros `setpar_par1.C`, `setpar_par2.C` etc. The macros have no arguments, e.g. `setpar_par1()`.

4.5.7 Go4 conditions

Conditions are objects holding window limits or polygons. One or two values can be checked against the limits or the polygon, respectively. In addition the conditions have test and true counters. They can be set to return always true or false or return the inverted test result. They can be edited by the GUI (see chapter 6.8.2, page 65). They can be used to steer the analysis flow. They are saved/retrieved to/from the auto-save file, if enabled. They can be edited in the auto-save file. In the editor they can be saved/retrieved to/from files. If a mechanism like for the parameters (4) is wanted, one can use macro `savecond.C([file],wildcard,prefix)` from `$GO4SYS/macros` creating macros (one per condition) to set all conditions to their current values,. The names are built from prefix and condition name. The macro can be executed in CINT (then the conditions are taken from a file), or in the GUI or in the analysis.
Example:
```
root[0].x savecond.C("myfile.root","*","setcon")
```
would produce macros `setcon_cond1.C`, `setcon_cond2.C` etc. The macros have three arguments: restore flags, restore counters, reset counters (0=no, 1=yes), e.g. `setcon_cond1(1,0,1)`.

4.5.8 Start-up of the analysis slave

When starting the Go4 analysis in GUI mode, the following actions take place in that order:

1. The **Launch Analysis** GUI panel started by  reads some settings from file `$GO4SYS/Go4Library/Go4LaunchClientPrefs.txt` and invokes shell script `$GO4SYS/Go4Library/Go4ClientStartup.ksh` (or if GUI was started in `-client` mode `$GO4SYS/Go4Library/Go4ServerStartup.ksh`, respectively). Both scripts invoke script `AnalysisStart.sh`
   in the current directory of the user analysis. Here the user may add his own initializations.
2. In **"as client"** mode (default), the executable **`MainUserAnalysis`** is started in GUI mode with the parameters:
   `MainUserAnalysis -gui <analysis name> <host name> <port number>`
   The parameters analysis name and host name are taken from the launch client GUI panel, the port number is dynamical (displayed on GUI start-up in the shell, often=5000). Instead of launching the client from the GUI, one may start the analysis client manually from a separate shell (do not forget "`. go4login`" !) with the matching host name and port number for the running Go4 GUI. This can be useful if the analysis shall run under `gdb`, or if `ssh/rsh` fails for some reasons.
   Started in **"as server"** mode, the analysis executable (e.g. in Go4Example2Step) is called with arguments parameters
   `MainUserAnalysis -server <analysis name>`
   and starts in server mode (see also 6.3.2, page 46).
3. ***TGo4Analysis*** is created and initializes the analysis framework. Then the constructor of the user subclass (e.g. *TXXXAnalysis*) defines the list of analysis steps with initial event parameters (input and output filenames) and auto-save settings just as passed from the **`MainUserAnalysis`**. Additionally, some user objects may be created and registered here. **Note that histograms registered here are never updated or replaced from the Go4 auto-save file and exist only until the analysis client is terminated. Conditions and parameters, however, are updated when the auto-save file is loaded and if their name is existing there**.
4. The analysis slave, if in client mode, connects to the Go4 GUI**.** Optionally, the Go4 histogram/object server is created.
   Note that the analysis in server mode does not connect automatically to the starting GUI, but waits for a separate connect request with login and password from any GUI. Only after this explicit connection the GUI in `-client` mode gets control over the analysis server!
5. The analysis settings are loaded from the default preferences file `Go4AnalysisPrefs.root`. A message is sent to the GUI: "**`Analysis Client MyClient: Status Loaded from file Go4AnalysisPrefs.root`** " (if successful). Note that all settings specified before in the compiled code (auto-save file name, event sources, etc.) are overwritten if the preferences file exists.
6. The analysis objects are loaded and updated from the auto-save file. The file name from the loaded analysis settings is used, if existing. Otherwise, the filename specified in the preceding user code by *SetAutoSaveFile(Text_t* name)* is used. If successful, a message is sent to the GUI: *"***`Analysis Client MyClient: Objects Loaded.`***"* If auto-saving was disabled completely by calling *SetAutoSave(kFALSE*), the auto-save file is not opened here even if it exists, and no objects are loaded! **The "overwrite filename" option in the auto-save settings must be disabled** to recover objects of a previous auto-save file; otherwise, all objects in an old file of the same name are lost!

7. The analysis settings are displayed on the GUI. At this moment, the analysis configuration window pops up and shows the active settings. Note that for GUI client at analysis server, configuration does not pop up automatically after login, but has to be requested by "arrow right" button of analysis configuration window.
8. End of analysis start-up. A message is sent to the GUI: *"*`Analysis Client MyClient has finished initialization`*"*.
   Note that now the analysis itself is not yet initialized, i.e. the event objects have not been created, and there are still no connections to event sources, etc.

4.5.9 Submit settings and run analysis

At any time the user may apply new settings to the analysis and start/stop the run. Note that if GUI runs as client connected to an analysis server, these operations are permitted for controller or administrator login only. Here the following is happening in the described order:

1. **Submit** the analysis settings. The settings as displayed in the analysis configuration window are sent to the analysis client.
   i. First, an already existing analysis is closed (see below).
   ii. The analysis is initialized with the new settings. Objects are loaded from the new auto-save file except auto-save is disabled by *SetAutoSave(kFALSE*). The file name is as specified in the configuration window.
   iii. The event objects are created. Event sources and stores are opened. The constructors of all user events and event processors are executed. **Note that any object (histogram, parameter, etc) which is created and registered in the user event constructors might replace an object of same name that was loaded from the auto-save file before!** To continue working with the loaded objects, the user should request pointer to the object by name from the framework here. Only if the object was not found it should be created anew.

   After submit, the Analysis browser can be refreshed by . When an analysis was running before, the new analysis is started immediately and the refresh is done automatically.
2. **Start** the analysis with **:**
   i. The Go4 GUI will send the start command and refresh the view in the analysis browser.
   ii. The *UserPreLoop()* function is executed once. Here transient pointers to data might be initialized, values from a user file might be read, etc.
   iii. The Analysis event loop is starting. For each event the analysis steps, the dynamic list entries, and the *UserEventFunc()* are executed. The loop will run until the event source is at the end, an error occurs, or the stop command is applied by the user.
3. **Stop** the analysis with :
   i. The event loop is halted. This will not close the analysis itself, i.e. all event objects still exist, event sources and -stores are still open. When restarting the analysis by , it will continue with the next event.
   ii. The *UserPostLoop()* function is executed once. Here transient pointers should be reset to 0, user files might be written or closed, etc.
4. **Save** configuration settings: At any time the current settings can be saved to a preferences file. This will not affect the running analysis. Note that after changing the settings in the analysis configuration window they must first be submitted to save them!
5. **Load** Settings: Loading analysis settings from a preferences file will immediately close the running analysis. The closing actions are just as described below. However, the loaded settings are not initialized until they have been **Submitted** again from the analysis configuration window!

4.5.10 Shutdown of the analysis client

The analysis client is shut down with the a button. This will take the following actions

1. The connection between analysis and GUI is closed.
2. The destructor of the user analysis class is executed.
3. **Close** of the analysis:
   i. Objects are written to the previous auto-save file, if *SetAutoSave(kTRUE*).
   ii. The event objects are deleted. Go4 event sources (.lmd files and MBS connections) are closed. Event stores (.root files) are finally written and closed. The destructors of all user events and event processor classes are executed. All references to the event objects are deleted from the Go4 folders.
   iii. The dynamic list is reset. All pointers to non existing objects are cleaned up.

4. The analysis client executable terminates. The Go4 GUI is ready to connect the next analysis client.

4.5.11 Disconnect or shutdown analysis server

The GUI may disconnect the analysis server with the  button. **This will neither stop the analysis nor shutdown the server task, but just close the connections to this GUI.** Additionally, when connected to an analysis server, the GUI has a  button in the analysis toolbar and a menu for **Shutdown Analysis server**. This is permitted in administrator mode only! This will take the following actions:

1. Analysis server broadcasts message about shutdown to all GUI clients connected. The GUIs will cease monitoring activities and prepare for disconnect.
2. The destructor of the user analysis class is executed.
3. **Close** of the analysis, see details in 4.5.10
4. The analysis server disconnects all GUI clients fast, i.e. without handshaking protocol, and terminates.

5 Analysis Examples

To begin with Go4, there are examples of analysis packages at `$GO4SYS/Go4ExampleSimple`, `$GO4SYS/Go4Example1Step`, `$GO4SYS/Go4Example2Step` and `$GO4SYS/Go4ExampleMesh`. The differences are:

Example	Analysis	Step factories	Event objects	Steps
Simple	*TGo4Analysis*	*TGo4StepFactory*	*TGo4EventElement*	Analysis
1Step	*TXXXAnalysis*	*TGo4StepFactory*	*TXXXEvent*	Analysis
2Step	*TXXXAnalysis*	*TXXXUnpackFact* *TXXXAnlFact*	*TXXXUnpackEvent* *TXXXAnlEvent*	Unpack Analysis
Mesh	*TMeshAnalysis*	*TGo4StepFactory*		13 different

5.1 *Using the examples at GSI*

When using Go4 at GSI where it is already installed, Go4 is set up by

```
. go4login
```

Note that there must be a space behind the dot. To see all relevant environment variables use command

```
go4version
```

The output of this command would be helpful if you report problems.

5.2 *Prepare the packages*

Copy the content of the directory `$GO4SYS/Go4Example1(2)Step` to a private one. You can directly make and run the example.

The package consists of the following files besides the include and source files:

- `Readme.txt`
- `AnalysisStart.sh` is the standard startup script for analysis client One may add here definitions needed by the analysis.
- `Makefile`
- `Go4UserAnalysisLinkDef.h` contains ROOTCINT class pragma definitions
- `rename.sh` is a script to set up file/class names.

5.2.1 Rename files/classes

Before renaming the files, cleanup by command:

```
make clean
```

There is one string included in all class and file names: XXX. It is recommended to replace this by another string more specific. This is done by

```
./rename.sh "XXX" "myname"
```

Example:

```
> ./rename.sh "XXX" "Ship"
```

Note that "myname" will be part of all class and file names! **Hint**: do not use a string which is already in any file-name!

5.2.2 Make

Then rebuild the package by command

```
make all
```

Shared library `libGo4UserAnalysis.so` and executable `MainUserAnalysis` should be created.

5.2.3 Using the GUI with rsh or ssh

rsh

When the analysis program is started from the GUI, one can choose between rsh and ssh. For rsh shell, make sure that the file `.rhosts` exists and that it contains entries for the machine names you want to run the Go4 analysis client on. The file `.rhosts` could e.g. look like this:

```
node01
node02
localhost
```

Note: localhost should be listed here, since this is the Go4 default.

ssh

To use ssh one must create ssh keys. These keys are specific to the node where they are created:

```
cd ~/.ssh
ssh-keygen -d
```

answer questions by RET or yes

```
cp id_dsa.pub authorized_keys
```

Now login via ssh once to all nodes (including `localhost`) where you want to run Go4. Answer `yes` to continue. Then exit and try again. No more prompting should occur. Running Go4 (via GUI) on the node where the keys have been created one can use `localhost` as analysis node. On other nodes one must use the node name. When you are the first time on a machine, try to login via ssh to that machine using the node name. If there is a prompt, answer `yes` to continue, exit and retry. Only if ssh works without prompting you can run Go4 on that machine (via GUI).

5.3 *Simple example with one step*

This package on `Go4ExampleSimple` contains a simple running Go4 analysis. It contains one analysis step. It uses the standard Go4 analysis classes *TGo4Analysis*, *TGo4StepFactory and TGo4EventElement*. Therefore the functions *UserPreLoop(), UserPostLoop(),* and *UserEventFunc()* are not available. No data can be stored in the output event. The example uses some conditions and some parameter objects. The step is reading events from a standard MBS event source filling some histograms. No output file is written. The analysis processes up to eight long word values from up to two sub events. A suited input file can be found on the Go4 web or "MBS Random" event source can be used. All classes are defined and declared in two files (*.h and *.cxx). Additional descriptions are in the source files.

5.3.1 Main program and analysis

Main: **MainUserAnalysis**

The main program can be started from the Go4 GUI (see chapter 6.3, page 45) or by command line:

```
./MainUserAnalysis   -file|-trans|-stream|-evserv|-revserv   input   [-p   port]
[events]
./MainUserAnalysis -f myfile.lmd
./MainUserAnalysis -e MBS42 1000
```

The events can be read from standard GSI lmd files or MBS or event servers. For each event the user event processor *TXXXProc* (function *BuildEvent*) is called. This user event processor fills some histograms.

5.3.2 Main macro

The macro `MainUserAnalysisMacro.C` can run directly in ROOT. It needs a `.rootmap` file for automatically loading all necessary libraries. This file is created by the new files `Makefile` and `Module.mk` from the example.

5.3.3 Analysis step

The analysis, analysis factory, and analysis step (all standard Go4 classes) are created in the main program. The input is specified by a set of macros (`file.C, trans.C, stream.C, evserv.C, revserv.C)`. Other setups are done in macro `setup.C`. The macros are called in the main program by *gROOT->ProcessLine*.

The event filled: ***TGo4EventElement*** (no data, no user specific code)
The processor: ***TXXXProc***

The standard factory created in the main program keeps all information about the step. No user event class is used in this example. Members of *TXXXProc* are histogram, condition, and parameter pointers used in the event function *BuildEvent()*. In the constructor of *TXXXProc* the histograms, parameters and conditions are created. Method *BuildEvent()* - called event by event - gets the output event pointer as argument, but do not fill any data. The input event pointer is retrieved from the framework. In the first part, data from the raw input MBS event are copied to arrays of *TXXXProc*. Two sub-events (crate 1,2) are processed. Then the histograms are filled, the 2d one with polygon conditions.

5.3.4 Parameters

Parameter class ***TXXXParam***
In this class one can store parameters, and use them in all steps. Parameters can be modified from GUI.

5.3.5 Auto-save file mechanism

See also chapter 6.4.4, page 49. By default auto-save is enabled for batch, disabled with GUI. The name of the file is built from the input by
`<input>_AS.root`
If it is enabled all objects are saved into this ROOT file at the end of the event loop. At startup the auto-save file is read and all objects are restored from that file. When *TGo4Analysis* is created, the auto-save file is not yet loaded. Therefore the objects created here are overwritten by the objects from auto-save file (if any), except histograms. From GUI, objects are loaded from auto-save file when the **Submit** button is pressed. Note that histograms are not cleared. One can inspect the content of the auto-save file with the Go4 GUI. Note that appropriate user libraries should be loaded into GUI to access data from auto-save file (see chapter 6.2, page 45).

5.3.6 Example log file

All lines with **** are from the example classes.

```
> MainUserAnalysis –event MBS42 1000

**** Input MBS42 (-e)
     process 1000 events
     auto save file: MBS42_AS.root

GO4-***> Welcome to Go4 Analysis Framework Release v2.6-0 (build 20600) !  <GO4
GO4-***> Create factory Factory <GO4
GO4-***> Analysis: Added analysis step Analysis  <GO4
**** evserv.C: Create MBS event server input MBS42
**** setup.C: Setup analysis
**** Main: starting analysis in batch mode ...
GO4-***> Opening AutoSave file MBS42_AS.root , UPDATE mode  <GO4
GO4-***> Analysis LoadObjects: Loading from autosave file MBS42_AS.root   <GO4
GO4-***> AutoSave file MBS42_AS.root was closed.  <GO4
**** Factory: Create input event for MBS
**** Factory: Create event processor XXXProc
**** TXXXProc: Create instance XXXProc
**** TXXXProc: Restored histograms from autosave
**** TXXXProc: Restored conditions from autosave
**** TXXXProc: Restored pictures from autosave
**** Factory: Create output event XXXEvent
**** Event XXXEvent has source XXXProc class: TXXXProc
GO4-***> AnalysisStepManager  --  Initializing EventClasses done.  <GO4
GO4-***> Analysis BaseClass --  Initializing EventClasses done.  <GO4
GO4-***> Analysis Implicit Loop for 1000 cycles is starting...  <GO4
GO4-***> Analysis Implicit Loop has finished after 1000 cycles.  <GO4
GO4-***> Opening AutoSave file MBS42_AS.root , UPDATE mode  <GO4
GO4-***> AutoSave file MBS42_AS.root was closed.  <GO4
**** TXXXProc: Delete instance
GO4-***> Analysis Step Manager  --  Analysis Steps were closed.  <GO4
**** Main: Done!
```

5.3.7 Adapting the example

Creating a new class
Provide the definition and implementation files (.h and .cxx)
Add class in `Go4UserAnalysisLinkDef.h`
Then make all.

Most probably you will change *TXXXParam* to keep useful parameters.
Then definitely you will change *TXXXProc* to create your histograms, conditions, pictures, and finally write your analysis function *BuildEvent()*.
Before running `MainUserAnalysisMacro.C` in ROOT CINT it must be changed because all parameters are hard coded.

5.4 *Example with one step*

This package on `Go4Example1Step` contains a simple running Go4 analysis. It contains one analysis step. It uses the standard Go4 step factory *TGo4StepFactory*, but a user written *TXXXAnalysis*. In this class the functions *UserPreLoop(), UserPostLoop(),* and *UserEventFunc()* can be used. It uses some conditions and some parameter objects. The step is reading events from a standard MBS event source filling some histograms and an output event. The analysis processes up to eight long word values from up to two sub events. A suited input file can be found on the Go4 web. All classes are defined and declared in two files (*.h and *.cxx). Additional descriptions are in the source files.

5.4.1 Main program and analysis

Main: **MainUserAnalysis**
Setup: ***TXXXAnalysis***

The main program can be started from the Go4 GUI (see chapter 6.3, page 45) or by command line:

```
./MainUserAnalysis    -file|-trans|-stream|-evserv|-revserv    input    [-output]
[events]
./MainUserAnalysis -f myfile.lmd
./MainUserAnalysis -e MBS42 1000
```

The events can be read from standard GSI lmd files or MBS or event servers. For each event the user event processor *TXXXProc* (method *BuildEvent()*) is called. This user event processor fills some histograms and an output event *TXXXEvent* (raw event) from the input event. The output events can optionally be stored in ROOT files. When a ROOT file with raw events exists, it can be viewed by the Go4 GUI using the tree viewer. Note that appropriate library should be loaded into GUI to let the viewer know *TXXXEvent* (see chapter 6.2, page 45).

5.4.2 Analysis step

In *TXXXAnalysis* the analysis step is created with the step factory and input and output parameters. Here the defaults are set concerning the event IO. Two parameter objects are created (*TXXXParam* and *TXXXControl*).

The event filled: ***TXXXEvent***
The processor: ***TXXXProc***

The standard factory created in *TXXXAnalysis* keeps all information about the step. The *TXXXEvent* contains the data members to be filled in *TXXXProc* from the input event (MBS 10,1). The *Clear()* method must clear all these members (an array for each crate in the example). The analysis code is in the event processor *TXXXProc*. Members are histograms, conditions, and parameter pointers used in the event function *BuildEvent()*. In the constructor of *TXXXProc* the histograms and conditions are created, and the pointers to the parameter objects (created in *TXXXAnalysis*) are retrieved. Function *BuildEvent* ()- called event by event - gets the output event pointer as argument (*TXXXEvent*). The input event pointer is retrieved from the framework. In the first part, data from the raw input MBS event are copied to the members of output event *TXXXEvent*. Two sub-events (crate 1,2) are processed. Then the histograms are filled, the 2d one with polygon conditions.
The name of the optional output file is built from the input by
`<input>_XXXEvent.root`

5.4.3 Parameters

Parameter class ***TXXXParam***
In this class one can store parameters, and use them in all steps. Parameters can be modified from GUI.
Parameter class ***TXXXControl***
This class has one member "fill" which is checked in *TXXXProc->Event()* to fill histograms or not. The macro `setfill.C(n)`, n=0,1 can be used in the GUI to switch the filling on or off. It creates macro `histofill.C()` which is actually used to set filling on or off (in *TXXXProc*). You can also modify `histofill.C` by editor before running the analysis.

5.4.4 Auto-save file mechanism

See also chapter 6.4.4, page 49. By default auto-save is enabled for batch, disabled with GUI. The name of the file is built from the input by
`<input>_AS.root`

If it is enabled all objects are saved into this ROOT file at the end of the event loop. At startup the auto-save file is read and all objects are restored from that file. When *TXXXAnalysis* is created, the auto-save file is not yet loaded. Therefore the objects created here are overwritten by the objects from auto-save file (if any), except histograms. From GUI, objects are loaded from auto-save file when the **Submit** button is pressed. Note that histograms are not cleared. One can inspect the content of the auto-save file with the Go4 GUI. Note that appropriate user libraries should be loaded into GUI to access data from auto-save file (see chapter 6.2, page 45).

5.4.5 Example log file

All lines with **** are from the example classes.

```
> MainUserAnalysis -e MBS42 1000


**** Input MBS42 (-e)
     output MBS42_XXXEvent.root disabled
     process 1000 events
     auto save file: MBS42_AS.root

GO4-***> Welcome to Go4 Analysis Framework Release v2.6-0 (build 20600) !  <GO4
GO4-***> Create factory Factory <GO4
**** Analysis: Create MBS event server input MBS42
GO4-***> Analysis: Added analysis step Analysis  <GO4
**** Main: starting analysis in batch mode ...
GO4-***> Opening AutoSave file MBS42_AS.root , UPDATE mode  <GO4
GO4-***> Analysis LoadObjects: Loading from autosave file MBS42_AS.root   <GO4
**** TXXXParam Par1 updated from auto save file
**** TXXXControl Control updated from auto save file
**** TXXXControl: Histogram filling enabled
GO4-***> AutoSave file MBS42_AS.root was closed.  <GO4
**** Factory: Create input event for MBS
**** Factory: Create event processor XXXProc
**** TXXXProc: Create instance XXXProc
**** TXXXControl: Histogram filling enabled
**** TXXXProc: Restored histograms from autosave
**** TXXXProc: Restored conditions from autosave
**** TXXXProc: Restored pictures from autosave
**** Factory: Create output event XXXEvent
**** TXXXEvent: Create instance XXXEvent
GO4-***> AnalysisStepManager  --  Initializing EventClasses done.  <GO4
GO4-***> Analysis BaseClass --  Initializing EventClasses done.  <GO4
**** TXXXAnalysis: PreLoop
GO4-***> Analysis Implicit Loop for 1000 cycles is starting...  <GO4
First event #: -1926055269
GO4-***> Analysis Implicit Loop has finished after 1000 cycles.  <GO4
**** TXXXAnalysis: PostLoop
Last event  #: -1926053525 Total events: 1000
**** TXXXAnalysis: Delete instance
GO4-***> Opening AutoSave file MBS42_AS.root , UPDATE mode  <GO4
GO4-***> AutoSave file MBS42_AS.root was closed.  <GO4
**** TXXXEvent: Delete instance
**** TXXXProc: Delete instance
GO4-***> Analysis Step Manager  --  Analysis Steps were closed.  <GO4
**** Main: Done!
```

5.4.6 Adapting the example

Creating a new class
Provide the definition and implementation files (.h and .cxx)
Add class in Go4UserAnalysisLinkDef.h
Then make all.

Most probably you will change *TXXXParam* to keep useful parameters.
Then you might change *TXXXEvent* to represent your event data.
Keep the *Clear()* method consistent with the data members!
Then definitely you will change *TXXXProc* to create your histograms, conditions, pictures, and finally write your analysis function *BuildEvent()*.

In *TXXXAnalysis* there are three more functions which eventually can be useful:
UserPreLoop () - called before event loop starts,
UserEventFunc() - called after each *TXXXProc::BuildEvent()*,
UserPostLoop () - called after event loop stopped.

5.5 *Example with two steps*

5.5.1 Main program and analysis:

This example on `Go4Example2Step` contains an unpack step and an analysis step. It uses some conditions and some parameter objects. Step one is reading events from a standard MBS event source filling some histograms and an output event. Step two uses this event as input and fills another output event and some more histograms. The analysis processes up to eight long word values from up to two sub events. A suited input file can be found on the Go4 web.
The main program (`MainUserAnalysis`) can be started from the Go4 GUI (see chapter 6.3, page 45) or by command line:

```
./MainUserAnalysis -file|-trans|-stream|-evserv|-revserv|-random
                    input [-server] [-port #][-output] [events]
./MainUserAnalysis -f myfile.lmd
./MainUserAnalysis -e MBS42 1000
```

The events are read from standard GSI event sources (in the GUI one can switch to MBS or event servers). Then the first user event processor is called (**Unpack**). This user event processor fills some histograms and the first user event (unpacked event) from MBS input event. Then the second user event processor is called (**Analysis**). This user event processor fills some other histograms and the second user event (calibrated event) from the first event. The events from the first and second step can optionally be stored in ROOT files (from GUI). When a ROOT file with unpacked events exists, the first step can be disabled, and this file can be selected as input for the second step (from GUI).
The main program builds the files names needed and creates the ***TXXXAnalysis***. Then it either connects to the GUI (when started from GUI) or starts the event loop (when started from shell). If `-server` is specified, GUIs invoked by `go4 -client` may connect.
Files created by the example are
Go4AnalysisPrefs.root: saved preferences
<input>_AS.root: auto-save file
<input>_XXXUnpack.root: event tree output from step 1
<input>_XXXAnl.root: event tree output from step 2

All classes are defined and declared in two files (*.h and *.cxx)
In *TXXXAnalysis* the two steps are created with their factories and input and output parameters. Here the defaults are set concerning the event IO. Two parameter objects are created (*TXXXParameter*). They can be used in both steps.

5.5.2 Step one: unpack

The factory: ***TXXXUnpackFact***
The event filled: ***TXXXUnpackEvent***
The processor: ***TXXXUnpackProc***

The factory *TXXXUnpackFact* normally need not to be changed as long as standard GSI event sources are used.
The *TXXXUnpackEvent* contains the data members to be filled from the input event (MBS 10,1). Only the *Clear()* method must be changed to clear all these members.
The unpacking code is in the event processor *TXXXUnpackProc*. Members are histograms, conditions, and parameter pointers used in the event method *XXXUnpack()*. This name can be chosen by the user. In the *Fill()* method of *TXXXUnpackEvent* this method must be called. In the constructor of *TXXXUnpackProc* the histograms and conditions are created, and the pointers to the parameter objects (created in *TXXXAnalysis*) are set. *XXXUnpack()* - called event by event - gets the output event *TXXXUnpackEvent* as argument (`poutevt`). The input event is retrieved from the framework. The first eight channels of crate one and two are filled in histograms `Cr1Ch01-08`, `Cr2Ch01-08`, respectively. `His1g` is filled under condition `cHis1` on channel 0, `His2g` under condition `cHis2` on channel 1. When editing conditions `cHis1,2` histograms `His1,2` filled by channel 0,1 will be displayed automatically to set/display the condition values. Picture `condSet` shows histograms `His1,2` on top, `His1,2g` at bottom. Open the condition editor in the view panel of the picture. Conditions `cHis1,2` will be selectable. They are displayed in the pad where they should be set. Both conditions are attached to the picture (see chapter 6.8.4, page 67). Histogram `Cr1Ch1x2` is filled for three polygon conditions: `polycon, polyconar[0], polyconar[1]`, all on the same values as the histogram.

5.5.3 Step two: analysis

The factory: ***TXXXAnlFact***
The event filled: ***TXXXAnlEvent***
The processor: ***TXXXAnlProc***

The step two is build in the same way as step one.
Note that the *TXXXUnpackEvent* is used two times: once as output of step one, and once as input of step two. Therefore the *Fill()* method checks if *TXXXUnpackEvent* has to be filled by *XXXUnpack()* in step one or retrieved from input file of step two which should be an output file of step one. Step one must be disabled in the second case. The user method *XXXEventAnalysis()* always gets the pointer to the correct event. Histogram `Sum1` is filled by first 4 channels of crate 1 and first 4 channels of crate 2. All channels are gated with condition `wincon1`. Histograms `Sum2,3` are filled similar, but without gate, and shifted by `XXXPar1,2->frP1`. Histogram `Sum1calib` is filled like `Sum1` without gate but with values calibrated by method *TXXXCalibPar->Energy()* of parameter `calipar`.

5.5.4 Parameters

With the ***TXXXParameter*** class one can store parameters, and use them in all steps. Parameters can be modified from GUI by double click.
In ***TXXXCalibPar*** is an example how to use fitters in parameters to calibrate histograms (more chapter 6.11.3, page 73).

5.5.5 Conditions

There are a few conditions created in *TXXXUnpackProc*. One (polycon) is used in *XXXUnpack()* for the accumulation of histogram Cr1Ch1x2. Another one (wincon1) is used in *XXXEventAnalysis()* of *TXXXAnlProc* to fill histogram Sum1. Conditions can be modified by double click in the browser. One can attach a histogram to a condition or attach conditions to picture pads to ensure that the condition is displayed/set on the proper display.

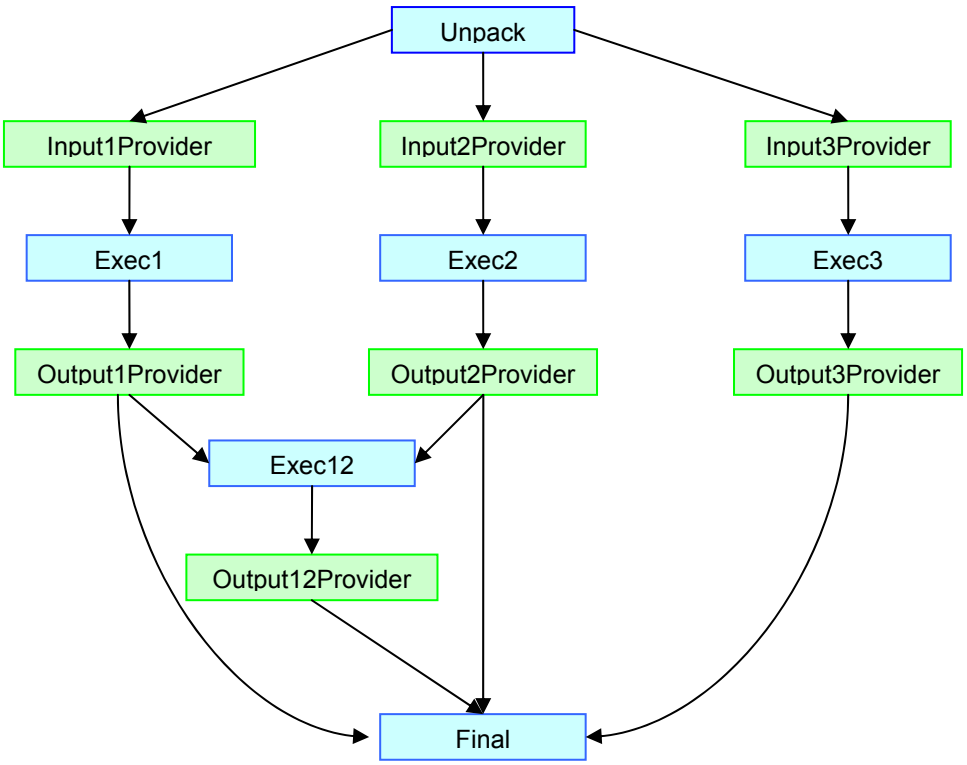
5.6 *Example of analysis mesh*

This example on `Go4ExampleMesh` shows how to set up a Go4 analysis of several steps that build a mesh of parallel analysis branches with different result generations. Additionally, one can see how the improved TGo4FileSource class supports partial input from a ROOT tree.

5.6.1 Structure:

The setup of the mesh analysis is done in the constructor of the *TMeshAnalysis* class. As in the Go4ExampleSimple, the general *TGo4StepFactory* is used to specify the event objects by name and class name. An overall of 13 analysis steps is defined for this example. Generally, the analysis mesh consists in two different kinds of steps, the execution steps and the provider steps. The unpack step, however, is as in the other examples just delivering sample data from a *TGo4MbsSource* (standard Go4 gauss example).

The step structure of the example mesh is as sketched in this figure (arrows show dataflow):



5.6.2 Execution steps:

These analysis steps do the actual analysis work, i.e. they convert some input event into the output event. This is the same as in the more simple examples (2-Step). However, to realize a mesh structure, the execution steps do not work directly on their own input event as assigned from the Go4 framework, but use the input event of one or more provider steps. The execution steps can access the input event pointers of any provider step by the provider step name, using the *GetInputEvent("stepname")* method. Note that the native input event of the execution steps is never used here (except for the very first "Unpack" step that processes the initial MBS event directly, without a provider step). There are no histogramming actions in the execution steps. To view the result data one has to use a dynamic list histogram or perform a *TTree::Draw* on the output event's tree, if existing.

5.6.3 Provider steps:

These analysis steps do not perform any analysis work at all, but only make sure that their own input event is always set correctly for the following execution steps, depending on the data flow situation. Generally, there are two cases:

- the provider step reads the input event directly from a branch of a ROOT tree (*TGo4FileSource*). In this case, the input event remains the native input event of this step as created in the step factory.
- the provider step refers to the result event of a previous execution step.

In this case, the provider processor itself has to find the correct event pointer by name from the Go4 object management. The default Go4 framework mechanism to handle these two cases will not suffice here, since it was designed for a subsequent order of steps and not for a mesh with parallel execution branches.
To do this job, all provider steps use the *TMeshProviderProc* class as general event processor, and the *TMeshDummyEvent* class as pseudo output event. The *TMeshDummyEvent* is necessary, because the Go4 framework will always call the *Fill()* method of the step's output event to execute any action of the step. So *TMeshDummyEvent::Fill()* calls method *TGo4ProviderProc::SetRealInput()* to set the pointer to the desired input event correctly.
If the input event is not read from file (native input event of this step), the provider processor has to search for it by name using the method *TGo4Analysis::GetEventStructure("name")*. However, the Go4 framework so far does not offer any additional parameter to specify the name of the appropriate input for a provider step. Therefore, this example uses the trick to derive the event name search string from the name of the provider processor itself: the name of this processor (up to the "_") is the name of the required event. Note that *TGo4StepFactory* forbids to use same names for different objects, since the object name is used as pointer name in the *ProcessLine()* call; therefore the processor name can not be identical with the input event name, but must differ by the "_" extension.
Additionally, the provider steps use the new partial input feature of the TGo4FileSource class (since Go4v2.9). The name of the event structure defines the name of the *TTree* branch that should be read from the input file. The first three provider steps use different parts of the *TMeshRawEvent* each. If the input event name is set to the name of the corresponding tree branch (e.g. "`RawEvent.fxSub1`"), the file source will only read this branch from the tree. If the input event name is set to the full name of the raw event ("`RawEvent`", commented out in this example), the complete event is streamed, including the not used parts. Note that in both cases the event object must consist in the full *TMeshRawEvent*, although in the partial input case only one sub-event is filled. This is required for a proper event reconstruction due to the ROOT TTree mechanism. In this example, the partial event input might increase the process speed by a factor of 2 compared to the full event input.

5.6.4 Configuration:

Although the step configuration can be defined as usual from the analysis configuration GUI, not all combinations of enabled and disabled steps make sense to process a subpart of the complete analysis mesh. For example, if execution step 2 shall be processed, the corresponding provider step for its input event has to be enabled, too. Note that the standard step consistency check of the Go4 framework is disabled here to run such a mesh at all (*SetStepChecking(kFALSE*)). So it is user responsibility to ensure that all required event objects are available for a certain setup. Moreover, with >13 analysis steps the standard analysis configuration GUI becomes quite inconvenient.
Therefore, the example uses a Go4 parameter *TMeshParameter* for the easy setup of the configuration. This parameter has just a set of boolean flags to determine which execution step shall be enabled. Depending on this setup, the *UpdateFrom()* method of the parameter also enables or disables the required provider steps. However, the parameter does not contain the full information of the input file names for the providers yet (In a "real" application, this could be implemented in a similar way though).
Thus the configuration procedure looks like this. The *TMeshParameter* is edited on the GUI to enable the desired execution steps. The parameter is send to analysis and switches the steps on and off. Then the analysis configuration GUI has to be refreshed by the user pressing button  to view the new setup. Here the user may change the names of the event sources for the provider steps, if necessary. After submitting these settings again from the configuration GUI, the mesh setup is ready. Note that once the mesh is configured in this way, the configuration can be stored completely in the analysis preferences and restored on the next startup.
One could also think of a user defined GUI that handles both the setup of the *TMeshParameter*, and the rest of the analysis configuration in one window. This would offer the additional advantage that it could show the structure of the analysis mesh in a graphical way. However, such a user GUI is not delivered here, but can be created according to the hints given in package Go4UserGUI (see chapter 6.16, page 78).

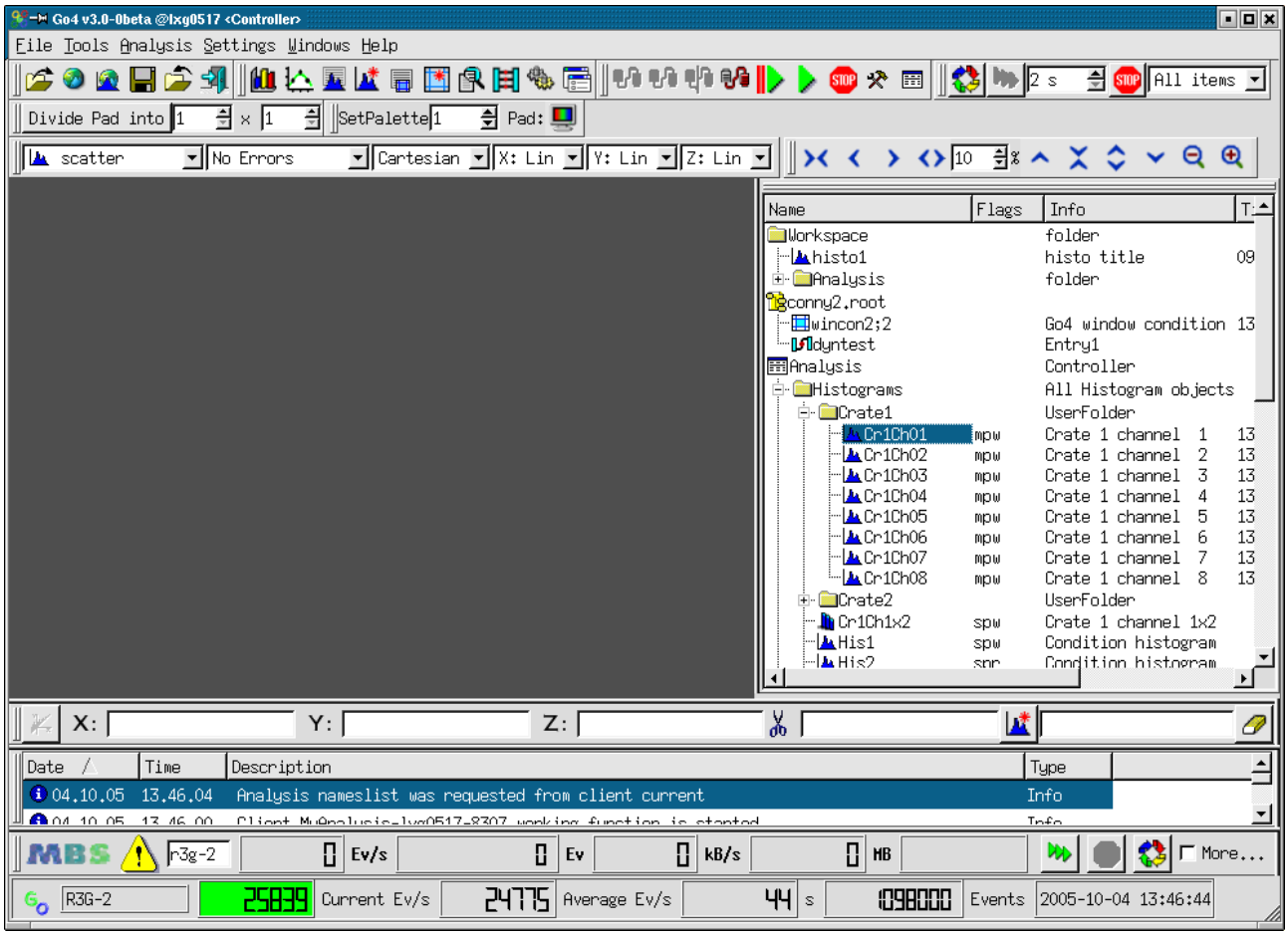
5.6.5 Usage of the example:

One way to test the example could look like this:

- Enable the first unpack step, disable the rest of the mesh. Use *TGo4MbsRandom* as event source for the Unpack and fill the output event *TMeshRawEvent* into a ROOT tree (switch on *TGo4FileStore* of unpack step). Do this until a reasonable number of events are processed.
- Disable the unpack step, enable one or more of the subsequent execution steps. The input for the first 3 provider steps should be the ROOT file that was produced before. Note that the first providers could also read their sub-events from different files. Eventually, produce further output trees from the execution steps.
- Change the setup in a way that only one branch of the mesh is processed, e.g. only `Exec3` and `Final`.
- Change the setup in a way that only a certain generation of events is processed, e.g. only `Exec1`, `Exec2`, and `Exec3`, writing output files of their results. Alternatively, let only `Exec12` and `Final` work, reading their provider inputs from these output files.
- Change the example code and recompile to add another execution branch, e.g. with new steps for InputProvider4, Exec4, OutputProvider4, and collect the results in the existing final step. New classes *TMeshB4InputEvent*, *TMeshB4AnlProc*, and *TMeshB4OutputEvent* should be defined for this (these can be derived from the corresponding classes as existing for the `Exec3` branch).
- Create a new mesh analysis from this template that matches your analysis structure.

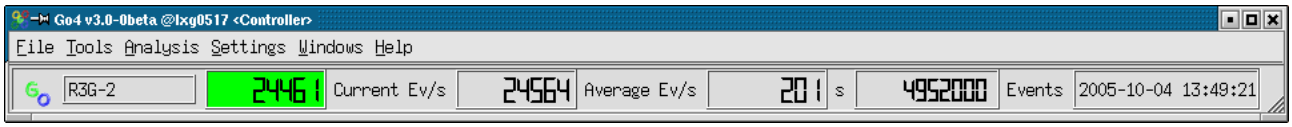
6 How to Use the Go4 GUI

The following picture shows the GUI with all elements. On the right side you see the Go4 browser. The left side will be the display panel. Below is the Tree viewer, and under this the message window, the mbs monitor, and the analysis status display. With **Show/Hide** in the **Settings** one can configure the layout and save/restore it. All buttons in the top row are also available as pull down menus commands.



gui300

This would be the minimal look of a running analysis (the date is updated from the analysis):



gui301

There are several screen movies on the Go4 web showing the use of the Go4 GUI.
There are many **keyboard shortcuts** to handle windows and actions. See chapter 11, page 90.

6.1 *GUI menus*

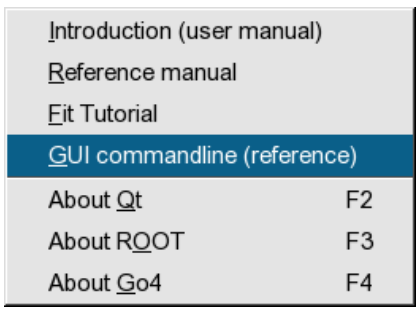
The icons in the top line are grouped into three segments corresponding to the first three pull down menus **File, Tools**, and **Analysis**.

6.1.1 File, Tools, Analysis menus

Pull down	Icon	Function
File		**Open:** opens local ROOT file
		Open Remote: open *TNetFile*, *TWebFile* or *TRFIOFile* to access remote data
		Open HServer: open conntection to gsi histogram server
		Save memory: save content of the memory browser into a ROOT file
		Close all files: close all ROOT files opened in file browser
		Exit: closes window and exit from GUI
Tools		**View Panel:** creates window (canvas) to display histogram(s)
		Fit Panel: opens fit panel
		Histogram properties: opens window showing histogram properties
		Create New His: opens histogram creation window
		Condition properties: opens window showing conditions properties
		Condition Editor: opens central condition editor
		Event Printout: examine current event contents
		Create Dyn. List Entry: histogramming on the fly
		Load Libraries: opens tool to load ROOT libraries
		User GUI: starts user GUI
Analysis		**Launch Analysis:** starts up the analysis task (as client or server)
		Connect to Analysis: login to running analysis server
		Prepare connection: allow external analysis client connect to this gui
		Disconnect Analysis: remove connection without analysis server shutdown.
		Shutdown Analysis server: in administrator mode only!
		Set+Start: submit setting and start analysis
		Start: start analysis events loop (after setup and submit)
		Stop: stop analysis events loop
		Configuration: open the configuration windows
		Analysis Window: opens the output window of the analysis

6.1.2 Help menu

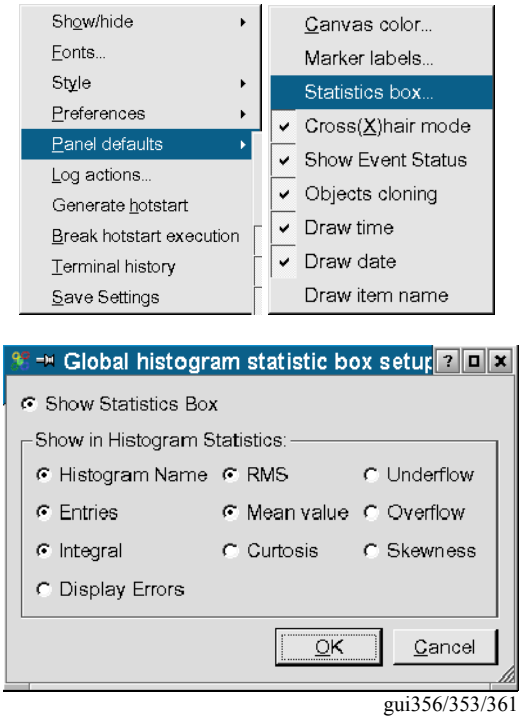
The help menu provides manuals on-line.



gui357

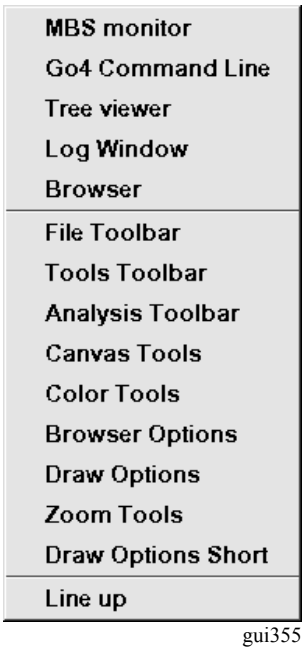
6.1.3 Settings menu

In the **Settings** pull down menu as shown on the left side one can set **Fonts** and **Style, Preferences** and **Panel defaults,** where one can set the histogram **Statistics box** and view panel layout. **Preferences** specifies when objects are fetched automatically from analysis.
You can adjust all fields according your needs. Then **Save Settings**. The next start of the GUI will restore the saved layout. Note that settings also contain other preferences, like window geometry and tools visibility, view panel background color and crosshair mode, graphical marker appearance, connection setup parameters, etc. By default, the settings are stored in text files `./.go4/go4localrc` and `./.go4/go4toolsrc`. To get the standard setup one may delete these two files. If the current directory does not contain a Go4 settings file on Go4 GUI startup, it will be created using the global account preferences at `$HOME/.qt`, or from the standard installation settings.
Settings behavior can be changed using environment variable `GO4SETTINGS`. If this is set, the GUI preferences are used from directory `$GO4SETTINGS`. If `GO4SETTINGS` contains keyword `ACCOUNT`, the Go4 settings at `$HOME/.qt` are used (like in previous Go4 versions).



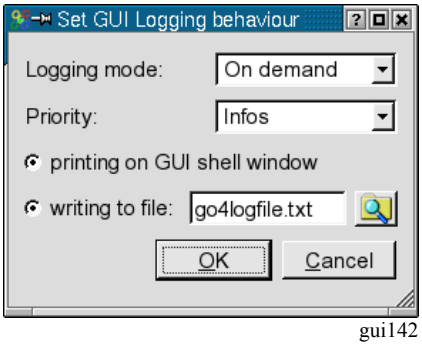
gui356/353/361

With the **Show/hide** entry of the settings menu (or with RMB in an empty menu region) one gets the window on the right to select which tools shall be visible. The actual content of these windows is preserved even if they are not displayed. This is also available as popup menu when clicking the right mouse button on an empty field of the main window.



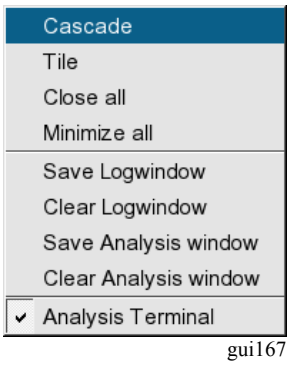
gui355

The **Log actions** of the GUI can be defined in a setup window from the settings menu. By default, the log output (e.g. condition properties, histogram information) is printed into the shell window where the GUI was started from. Additionally, a text file may be specified for output. **Logging mode** specifies if log output is produced **On demand** only (i.e. on clicking the log button when available), or **Automatic** whenever the content of an editor/information window changes. **Priority** defines the level of output suppression: **Errors, Warnings, Infos,** or **Debugs**. Level **Errors** will only log in case of an error, **Debug**s will printout even debug information of the Go4 kernel. This reflects the priority of the *TGo4Log::Message()* method.



gui142

In the **Panel defaults►Canvas color** menu the default background color for all newly opened view panels can be set. This color may be saved together with the other settings. **Panel defaults►Marker labels** specifies the default label layout.
The **Crosshair mode** entry toggles the default crosshair cursor on/off for all newly opened view panels. This crosshair state may be saved together with the other settings. However, the crosshair can be switched independently for each pad in the menu of the view panel (see chapter 6.7, page 57).
The **Generate hotstart** entry will save the current state of the GUI (window geometry, objects in memory and monitoring list, objects in view panel, analysis settings) to a Go4 hot start file (`*.hotstart`). The name of the hot start file can be defined in file dialog here. When re-starting the Go4, the hot start file may be used as command line argument, restoring the state of GUI and analysis (see chapter 6.15, page 78).
With **Terminal history** the buffer size for the analysis output window can be limited.



gui167

6.1.4 Windows menu

The **Windows** pull down menu shown on the right side provides items to arrange the windows and to save and clear the analysis and log windows.

6.2 *Load libraries to GUI*

To access data from user defined classes (like parameters or events) a library including the ROOT dictionary is required. This library is produced by the make file and has the name `libGo4UserAnalysis.so`. It is recommended to load user libraries for non-Go4 classes (for instance, user event classes) before opening a file with a TTree, where object of these classes are stored. There are three different ways to do it.
First, any external shared library (with or without ROOT dictionary inside) can be loaded by press of the button on the main window. A file dialog then asks to specify the library to be loaded.
Second, set the environment variable `GO4USERLIBRARY` to a list of user libraries (separated by colons) to be loaded when the GUI starts. Typically before run the Go4 GUI the user should type in the shell:

```
export GO4USERLIBRARY=..../libGo4UserAnalysis.so:..../libOther.so
```

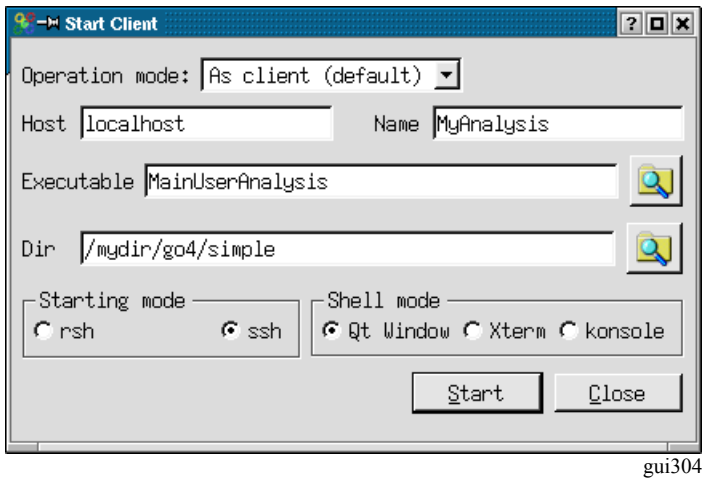
Third, the new possibility (since ROOT 4.00/08) for automatic load of libraries with a `.rootmap` file. This file contains information to automatically load all necessary libraries for user classes. All make files of the Go4 examples generate `.rootmap` files during compilation. To explicitly generate this file again, type `make map` after compilation. If this file is located in the current directory (where GUI is started) or in the user home directory, all libraries will be loaded automatically at the time when required. For more details about `.rootmap` files see the ROOT home page.

6.3 *Launch analysis*

Press the button (or **Alt a n** or **Strg n**). This will start the **Launch analysis** window to execute the analysis task on another host. The operation mode of the analysis task may be “As client” (default), or “As server”; this has to be specified in the selection box on top of the Start Anaylsis window.
The difference of these modes is that in client mode the analysis connects as client to the starting GUI and will be finished when the Go4 GUI terminates. There can be only one GUI connected to an analysis in client mode. The starting GUI will connect automatically to the analysis client after launching it with full controller priviliges.
In contrast to this, the analysis started “as server” will be an external process independent of the starting GUI. Therefore in server mode the analysis can not run embedded into the Qt Window of the GUI. Any number of Go4 GUIs may connect to this analysis server with different priviliges, but only one GUI may be the authorized controller. Especially the starting GUI has to login to the analysis server after launching it in a separate dialog window.

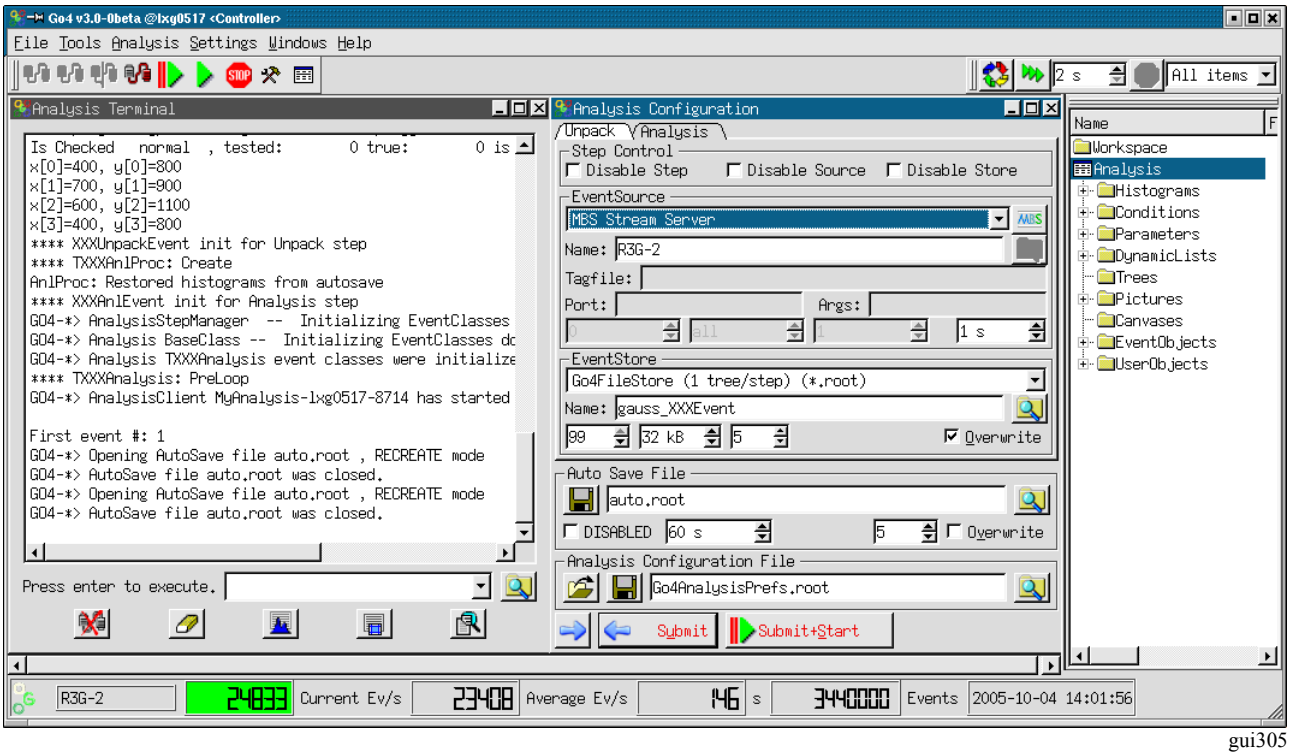
6.3.1 Launch analysis task in client mode

Besides the selection of the operation mode, the popup window expects an arbitrary name for the analysis and the node name of the machine where the client should be started. Normally this is the current node (`localhost`) as offered by default. Furthermore there are fields for the user working directory (in this directory the analysis is started) and the program name. **Note that these values are stored to, and retrieved from the current Go4 settings file.** Start the analysis with button **Start** or **RET**.



gui304

The client will be started by script `AnalysisStart.sh` in a remote shell (Starting mode rsh), or secure shell (Starting mode ssh). One should add in this script definitions needed by the analysis. The output is directed to a text window inside the GUI ("Qt Window"), or to an external Xterm, or to the KDE konsole (if existing), depending on the selected Shell mode.
After initialization the client connects to the GUI. When this procedure is done, the message "**`Starting analysis client ...Please wait`**" changes to "**`Editing Analysis Configuration ...`**" and the GUI is ready popping up an analysis terminal window and the analysis configuration window. Here the analysis steps can be configured (see chapter 6.4, page 48). Then the analysis must be set up by pressing **Submit** (or **Alt u**).
After setting up the analysis it is started by ▶ (or **Alt a s** or **Strg s**). In the browser the directory of the remote Analysis appears. The next figure shows the GUI with a running analysis. On the right side is the browser with the analysis directories; on the left side the analysis terminal, and the analysis configuration window.



gui305

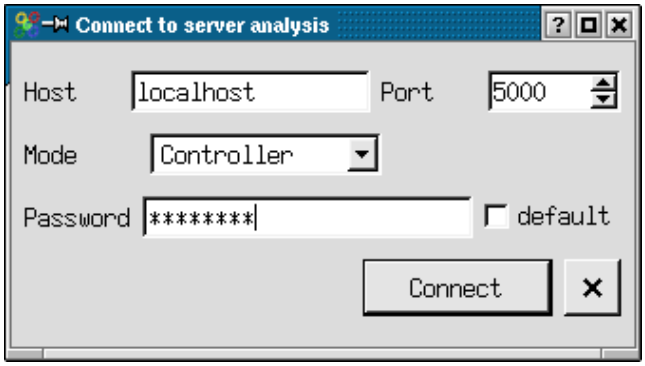
The configuration window is described in more detail in the next chapters.

6.3.2 Launch analysis task in server mode

To launch the analysis in server mode, the **Operation mode** in the start dialog window must be switched to "as server". The other settings are the same as described in section 6.3.1, except for the disabled possibility to run the analysis shell in the internal Qt window of the GUI. Immidieately after starting the analysis server, the "Connect to analysis server" dialog will pop up, expecting specifications for login of the GUI to the newly created server.

6.3.3 Connect to existing analysis server

Once the analysis server has been started (from the start dialog, from other GUI, or from external shell command line, respectively), one can connect this GUI to the server. This is done via the "Connect server" dialog that is available from the connect button in the analysis menu. If the server has been launched before from this GUI, the connect dialog will popup automatically.
Port number must match the connection port as printed out in server terminal window. **Host** should specify the node name of the server machine.
Three different accounts (roles) for login are provided: **Observer, Controller,** and **Administrator**. Each login has to be verified by a password. The Go4 default passwords **`go4view`** (observer), **`go4ctrl`** (controller), and **`go4super`** (administrator) are used when the **default** check box near the **Password** field is active. They may be changed in the `MainUserAnalysis` program by methods



gui306

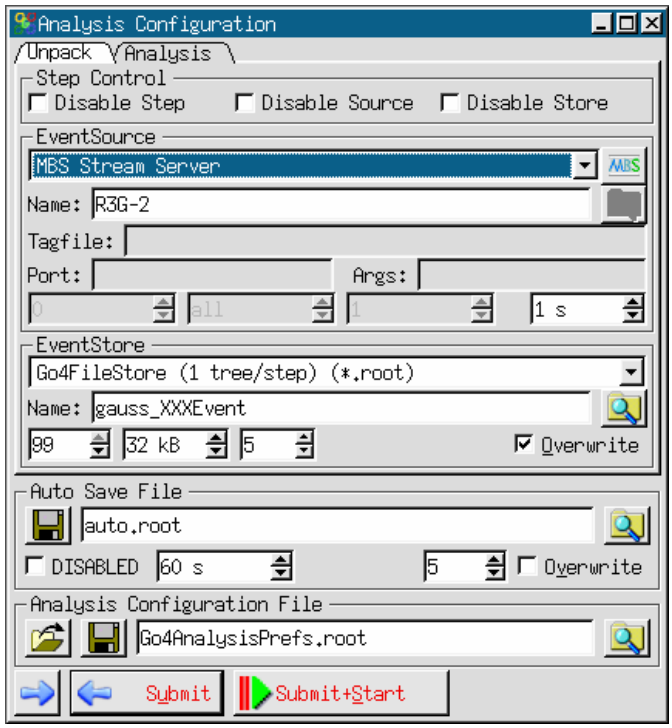
TGo4Analysis::SetObserverPassword(), *TGo4Analysis::SetControllerPassword()*, and *TGo4Analysis::SetAdministratorPassword()*, respectively. In the latter case, the correct password must be typed into the password field.
Only one controller or administrator may be logged in at the analysis server at the same time. If a controller (or administrator) GUI has already been attached, the next controller or administrator login will get an observer role. Observers may only view analysis objects and configuration, but may not modify them. **Submit**, **Start** and **Stop**, and remote macro execution is forbidden for observers, too. The controller account may modify all objects and the analysis setup and change the analysis running state, but may not shutdown the analysis server itself. Finally, only the administrator account may terminate the analysis server. After connection is established, the GUI main window title will show the role (**<Observer> , <CONTROLLER>, <ADMINISTRATOR>**).
When a controller after a connection wants to change the configuration he must open the Analysis Configuration window and get the configuration with . When the configuration is OK, submit. When connected as an observer, button may be used to get the object list from the analysis in the browser. You may also get the configuration, but cannot submit.
The GUI disconnects from the analysis by , but the analysis continues. To really shut down the analysis one has to use button (administrator only).

The `MainUserAnalysis` program must be adjusted to work as analysis server. The constructor of *TGo4AnalysisClient* got two additional arguments: `servermode` and `autorun`. The usage can be seen in `Go4Example2Step/MainUserAnalysis.cxx`. To run in server mode and controlled from GUI, `servermode` must be `kTRUE`. When started from the GUI, the analysis is started by `MainUserAnalysis -server name`.
In this case `servermode` is set `kTRUE` and `autorun kFALSE`.
Alternatively, the analysis can be started manually in server mode using the batch argument list with the first optional argument `-server`. This sets both `servermode` and `autorun kTRUE`. Then the analysis starts immediately using the setup specified by the arguments. Note that the preferences file is not used! This is useful when one wants to start `MainUserAnalysis` manually to be connected by GUI clients (on-line).

For analysis servers in ROOT macros see chapter 7, page 80

6.4 *Analysis controls*

6.4.1 Configuration window



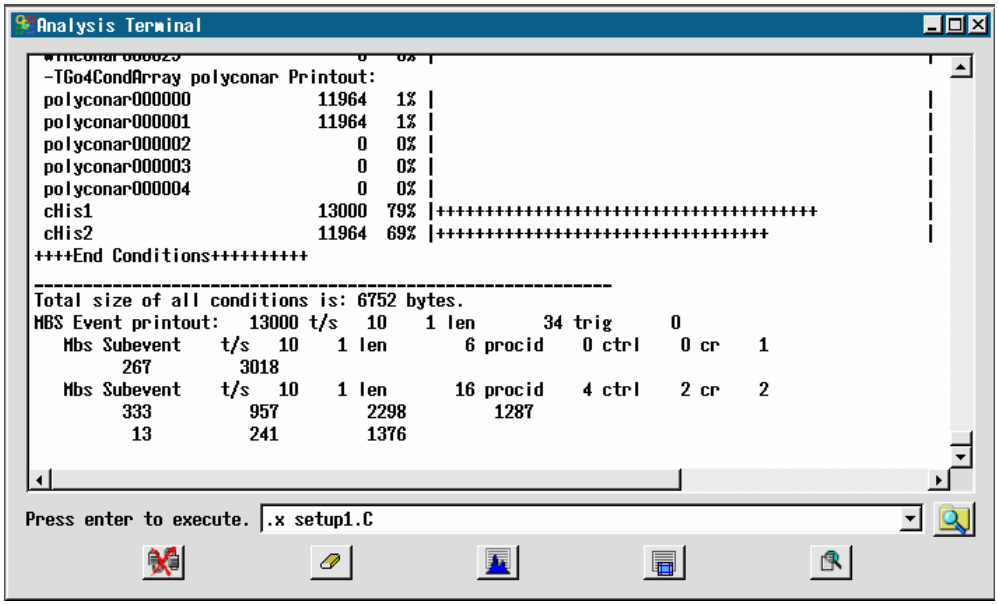
gui307

The Analysis configuration window shows the last valid setup of the analysis steps. These are taken from the user analysis constructor parameters, or from the ROOT file `Go4AnalysisPrefs.root` (in analysis working directory), if existing.
The Analysis configuration consists of the configuration parameters for each analysis step. The **analysis steps** are shown in different tabs of the configuration window. The values for event source, event store and working status of the analysis steps can be changed for each step separately. Depending on the chosen **Event Source**, different parameter fields will highlight for optional parameters. The MBS File, e.g., can specify an MBS **tag file** name (see MBS manual), and numbers for the **first** event, the **last** event and the event number **step** between subsequent events to be processed. Multiple input metafiles are supported by a preceding `@` character (see chapter 6.4.5, page 50). The Event Source **Remote Event Server** may need a **Port** number, other on-line sources can set the socket **timeout** in seconds. For user defined sources (see chapter 6.4.6, page 50), the optional string argument **Args** may be evaluated in the user step factory.
The **Event Store** settings define the ROOT **split** level and branch **buffer** size of the ROOT tree, and the file **compression** level. If the **Overwrite** radio button is false, new events will be appended to a previously written tree of the same event store name.
Moreover, steps may be **disabled** completely: the first step, e.g., can be left out and the second step may read its input from a previously created output file of the first step. **Note: the input of the actual first step must be specified; otherwise the analysis will not be initialized**!
The auto-save file for analysis objects (histograms, conditions, parameters, dynamic list connections) is defined for all steps with the auto-save **interval**, the file **compression** level, and the **Overwrite** option. Selecting **never** for the auto-save interval will prevent saving the objects during the analysis run. However, the auto-save file will be written at the analysis shutdown (resubmit next settings). Auto-saving can be disabled completely by the **DISABLED** checkbox, i.e. the auto-save file is not even opened for reading previous objects.
Note that the buttons at the different name fields will open a browser for the local file system to search for appropriate file names.
The new settings are activated on the analysis client by pressing the Submit button (or **Alt u**). **Note: you have to press Submit even if you want to apply the settings unchanged!** To synchronize the configuration window with the current analysis settings, the refresh button can be used. This is usually done automatically on first connection of the analysis, but it might be useful when starting the analysis manually from a different shell, or when changing the analysis setup independently from the GUI.
The Submit button closes the previous analysis (i.e. all files and connections will be closed, all event classes except for the analysis step factories will be deleted) and initializes the analysis with the new settings.
To have the changed settings available on the next analysis client startup, press the **Save** Button . This will write the current analysis settings to the file `Go4AnalysisPrefs.root` (default name for startup), or to any other ROOT file specified in the file dialog or the filename text field. Previously written configurations can be loaded using the **Load** button and the corresponding file dialog.

- **Note 1**: A changed **configuration must first be submitted** to the analysis before it can be saved.
- **Note 2**: When a new configuration is loaded, the previously active **analysis is closed without saving the configuration**. After loading a configuration it appears in the configuration window. **To initialize the analysis with these new settings, the submit button must be pressed!**

6.4.2 Analysis terminal window

When using the **Qt Window** option in the launch window, the analysis terminal window of the GUI shows all analysis printouts



gui133

Button clears the window, prints all histograms info, prints all conditions info (make window wide enough for the counter bars). Button will open the event information window (see chapter 6.14, page 77).
Additionally, it is possible to kill the analysis process with the button on the hard way. This will disconnect the analysis client after a while from the GUI and analysis can be launched again. However, this is not recommended since the ROOT output files may remain in a non valid state after the kill!
Analysis terminal output can be stored by **Windows►Save Analysis window** menu command to text file. Analysis terminal output history is limited by 100 Kbytes. This value can be changed in **Settings►Terminal history** menu command. To keep full history, 0 should be set.
When the analysis task is running in an external shell (xterm, konsole), the buttons and macro execution line will appear in a special dockwindow (see figure).

gui325

6.4.3 Macro execution in the analysis

The analysis terminal window offers the possibility to execute ROOT CINT commands and macros in the analysis task. Note that a history of previous commands of the session is available with the macro line combo box (mouse selection, or **arrow down** key). looks up for macro files.
Using the **go4** pointer (already set to *TGo4Analysis::Instance()*), one has access to all public methods of the analysis framework from inside the macro. Note that the shortcut @ exists here for *TGo4Analysis::Instance()->*, e.g. *@PrintHistograms("Cr1*")* will print all histograms with names matching the wildcard expression. In macros the environment variable **__GO4ANAMACRO__** is defined and may be checked. A detailed description can be found in the reference manual.
It is not necessary to load the Go4 libraries in the macro again, since these are known at runtime in the analysis anyway. See also macro execution in GUI (see 6.17, page 79). `$GO4SYS/macros` directory should be added to entry `Unix.*.Root.MacroPath` in `.rootrc` setup file.

6.4.4 Auto-save file mechanism

When auto-save is enabled (in MainUserAnalysis), all objects are saved into a ROOT file after every auto-save interval seconds time, and before termination. The auto-save file can also be written on demand by **Save** button in the configuration window. At startup of the analysis the following actions are done:

1. The analysis is created.
2. The auto-save file is read and all objects are restored from that file. Objects already existing, i.e. created in the analysis constructor, are overwritten by the objects from the auto-save file, except histograms. Existing histograms are not restored!
3. Before creating objects in the processor constructor or the *PreLoop()* method of the analysis one should check by the proper getter method if the object has been already restored from auto-save. If not, it can be created. If it is created while already existing the existing object is deleted first, i.e. the values from auto-save are lost.

When the analysis is controlled from GUI, objects are loaded from auto-save file when the **Submit** button is pressed (full sequence see chapter 4.5.9, page 28)

6.4.5 Multiple input files

There is the possibility to process multiple input files (source type MbsFile) in one analysis set-up. This can be achieved by wildcard characters in the **Event Source** name field, e.g. `*.lmd` or `data???_march03.lmd` or `*`. All files matching the wildcard expression will be read subsequently without closing the analysis; output events may be written into one event store. Additionally, one may specify the name of a metafile containing a list of inputs; the metafile name has to be preceded by an `@`, e.g. `@gaussfiles.lml`. Each line of the definition file `gaussfiles.lml` may contain the following format (values separated by blank spaces):

```
inputfile tagfile firstevent lastevent skipevents
```

The numbers of first and last event always refer to the running event count in the currently open event source, starting with number 1 each (not the event number inside the event header). The skip events number defines how many events shall be skipped in one file in between two processed events; this may be useful if a long term sample of a large input file shall be taken. The tag file may contain information which events shall be processed in the input file (see MBS manual).
At least the input file name must be specified; wildcards are not allowed here. Complete lines in the metafile may be commented out by a preceding "!" or "#" character.
Moreover, **metafile lines preceded by an `@` character are treated as ROOTCINT commands**, e.g.

```
@ .x setup.C
@ TGo4Analysis::Instance()->ShowEvent("Unpack");.
```

These commands are executed in between change of event source, thus allowing to use different setup parameters for different list-mode files.
Note that multiple input files also work in batch mode. However, wildcard expressions must be put in parentheses ("") if they are passed to the **`MainUserAnalysis`** as command line parameter. In batch mode the input file suffix is automatically expanded to `*.lmd`, if it was neither `.lmd` nor `.lml`. Therefore the meta file can also have suffix `.lmd`, i.e. `@myfiles` results in reading `myfiles.lmd` (although it is a plain text file). A better way is to use suffix `.lml`, because then one can omit the `@` and therefore the parentheses.

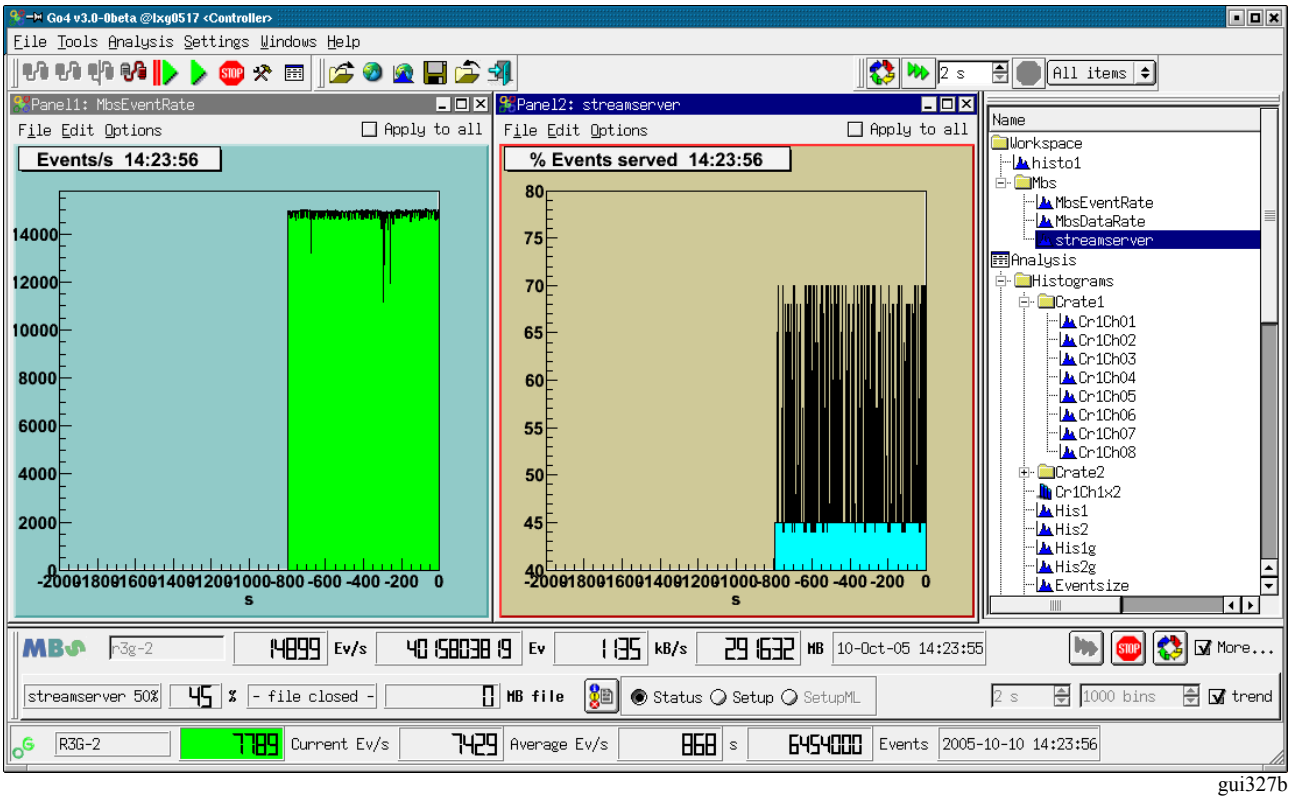
6.4.6 User defined event sources

Besides the delivered Go4 event sources for the standard MBS or ROOT file input, there is the possibility to define any other event source. In the analysis configuration window, there is the selection **UserSource** for the analysis step **Event Source** type. In this case, a ***TGo4UserSourceParameter*** object is passed to the step factory of the step. The user source name, and optionally, port number and a text argument can be specified in the configuration GUI to be evaluated on analysis initialization. Method *CreateEventSource()* must be re-implemented in the user step factory to react on a *TGo4UserSourceParameter* by creating any kind of *TGo4EventSource* subclass that the user had defined for his purpose. Note that method *CreateInputEvent()* should also be overwritten to create a raw event matching to the user event source, since the default of the base class *TGo4EventServerFactory* always delivers a *TGo4MbsEvent*.

The package **`Go4ExampleUserSource`** shows a simple example of a user defined event source reading data from an ASCII text file. Like the two step example, the package can be copied to a user working environment, and the class names can be renamed replacing the "TYYY-" prefix.
The event source class *TYYYEventSource* is prepared to handle any ASCII file containing columns of data separated by blank spaces. Each row is read and its values are converted in order into the Double_t fdData array of the raw event class *TYYYRawEvent*. The array expands automatically depending on the number of columns. Lines starting with "!" or "#" characters are treated as comments and are ignored. Thus these two classes need not to be modified for input of any ASCII files of that type. However, both the unpack procedure as specified in the event processor *TYYYUnpackProc* , and the unpack event class *TYYYUnpackEvent*, are depending on the column's meanings here and must be adjusted. Additional information can be found in the README.txt file of the example package.

6.4.7 MBS status monitor

When working with the gsi multi branch system mbs as event source, Go4 offers a monitoring tool that can request information from the status port of a running mbs system. This is available as dockwindow from the “Show/hide” menu, or will appear when the mbs button is pressed in the analysis configuration window.



gui327b

The screenshot shows the Go4 main window with the mbs monitor tool docked in the bottom part, right above the Go4 analysis status line. The mbs monitor by default shows just one line of information, but may be extended by the lower line with more details using the **More...** checkbox.

The upper line displays, from left to right: The mbs logo which is animated when the mbs acquisition is running; a text line to edit the mbs host name; event rate (events/s); total events acquired; data rate (kB/s); total data acquired (Mb); time and date of last refresh. On the right there are control buttons: With the mbs status server is newly connected and the information is refreshed. It is possible to refresh the status frequently, this is switched on and off with the buttons and , respectively. The refresh time can be chosen by the “seconds” spinbox in the lower line of the mbs monitor window.

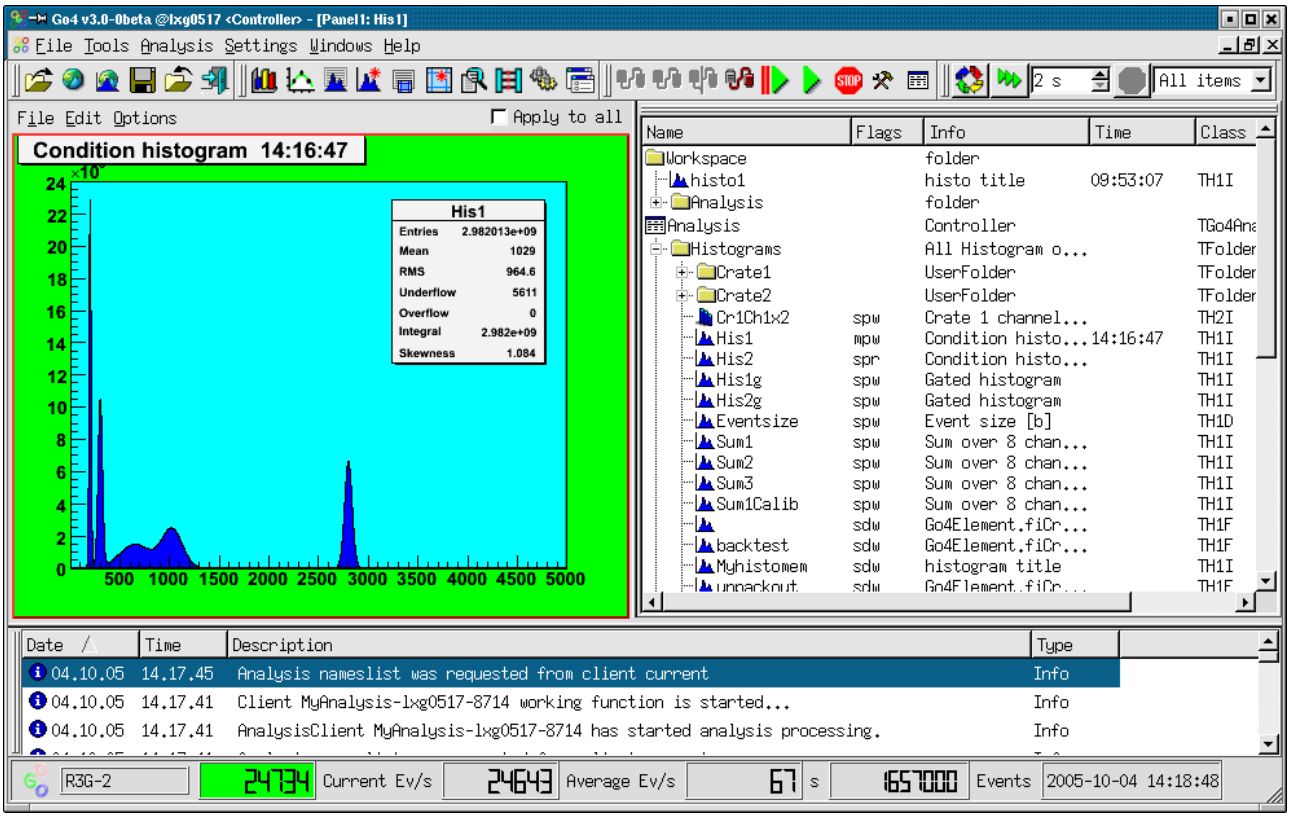
Additionally, the lower line displays (from left to right): Name of data server in use (streamserver, or eventserver) and percentage of delivered events 1/n, as it is set in the mbs by command `set stream n`, or `set event n`; percentage of *real* delivered events from this data server; name of the file which is currently written by the mbs, if existing, and total amount of data written to file since mbs startup. The button may be used to print the complete mbs status structure, the complete setup structure, or the multilayer setup structure, respectively, to the shell from which the gui was started. This is selected by the radiobuttons **Status**, **Setup**, and **SetupML**. Note that printout of multilayer setup is enabled only if a real multilayer setup exists in the observed mbs.

Besides the time selector for the monitoring frequency, the right side of the second line offers the possibility to switch on several **trending histograms**. This is done by the **trend** checkbox. The overall number of bins may be changed in the bins selector; the range of one histogram bin equals the monitoring frequency. Note that trending histograms are only written if the mbs status monitoring is turned on (i.e. no new entry in trend histogram by manual refresh using button). Three different trending histograms are currently produced: for the event rate, the data rate, and the percentage of delivered events at the mbs data server (streamserver or eventserver). They appear in the Go4 browser in the **Workspace/Mbs** folder and may be observered in Go4 view panels. The screenshot shows the trending histograms for event rate and streamserver event ratio.

Note that a warning sign will appear in the upper line if connection to mbs status server fails.

6.5 *The Go4 browser*

After pressing the analysis starts and the rates are displayed at the bottom as shown in the screen shot below. The analysis output window and the configuration window have been closed. A view panel created by has been opened and a histogram is displayed by dragging & dropping a histogram from the browser into the canvas. Note the logging window displaying messages from the remote analysis. This log panel can be opened in the **Settings** menu bar. The complete logging history may be saved into a text file by the **Windows►Save Logwindow** menu command.

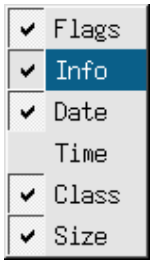


gui309

The Go4 browser on the right side shows objects from different data locations in a folder structure. Remote objects in the connected analysis task are listed under the **Analysis** branch. The **Workspace** folder contains all objects that are put into the memory of the local GUI, e.g. by creating fix copies of remote analysis objects. A root file opened from the files toolbar with the button will appear in a folder of the filename; similarly, a connection to a remote data source like the xrootd, the root webfile, or the gsi histogram server, shows up as separate browser branch.

6.5.1 Browser columns

Beside the "names" column showing the objects in their folder structure by symbols, the Go4 browser has configurable columns to display different kinds of properties of the displayed objects: Flags, Info, Date, Time, Class, and Size. These can be switched on and off by the menu that pops up on right mouse button click in one of these. Moreover, the order of these columns can be freely arranged in the browser by dragging and dropping their caption to a new position.



gui312

The **Flags** column will indicate certain properties of the object by letters:

- **m** - shall be monitored frequently; or **s** - is static until explicitely refreshed
- **d** - object may be deleted; or **p** - is protected against deletion
- **r** – read only, can not be reset; or **w** – writable, may be reset

The **Info** field will usually show the type of the folder, or the title of the ROOT object.
Date and **Time** columns show the date or time of the last object refresh to the GUI inernal cache (for remote data sources), or of the object creation (for local workspace), respectively.
Class column shows the class name, and **Size** will give an overall object size in bytes.

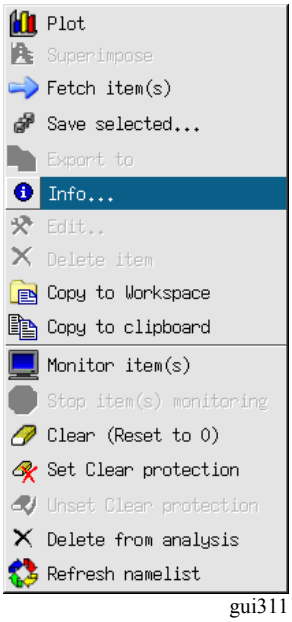
Browser

Name	Flags	Info	Date	Time	Class	Size
Analysis		Controller			TGo4Analys...	= 692068
Histograms		All Histogram objects	2005-10-04	14:24:51	TFolder	= 686280
Conditions		All Condition objects			TFolder	= 1456
Subfolder		UserFolder			TFolder	= 252
wincon1	spw	Go4 window condition	2005-10-04	14:24:51	TGo4WinCond	164
wincon2	spw	Go4 window condition	2005-10-04	14:24:51	TGo4WinCond	164
polycon	spw	Go4 polygon condition	2005-10-04	14:24:51	TGo4PolyCond	120
winconar	spw	TGo4WinCond	2005-10-04	14:24:51	TGo4CondArray	132
polyconar	spw	TGo4PolyCond	2005-10-04	14:24:51	TGo4CondArray	132
cHis1	spw	Go4 window condition	2005-10-04	14:24:51	TGo4WinCond	164
cHis2	spw	Go4 window condition	2005-10-04	14:24:51	TGo4WinCond	164
myConny	sdw	1-D window condition	2005-10-04	14:24:51	TGo4WinCond	164
Parameters		All Parameter objects			TFolder	= 2328
XXXPar1		This is a Go4 Paramete...			TXXXParameter	920
XXXPar2		This is a Go4 Paramete...			TXXXParameter	920
sizefitter		This is a Go4 Paramete...			TGo4Fitter...	32
specfitter		This is a Go4 Paramete...			TGo4Fitter...	32
CaliPar		This is a Go4 Paramete...			TXXXCalibPar	424
DynamicLists		Dynamic List Instances			TFolder	= 884
Pictures		Picture objects			TFolder	= 184
condSet	spw	Set conditions	2005-10-04	14:24:51	TGo4Picture	92
Picture1	spw	Picture example	2005-10-04	14:24:51	TGo4Picture	92
Canvases		All TCanvases			TFolder	
UserObjects		For User Objects			TFolder	= 156
Calibration	spw		2005-10-04	14:24:51	TGraph	100
MultiTest	spw	This is a test multigraph	2005-10-04	14:24:51	TMultiGraph	56
Trees		References to trees			TFolder	
AnalysisxTree		This is a Go4 Status O...			TTree	
XXXAnlEvent.		XXXAnlEvent.			TFolder	
XXXAnlEvent.TGo4Event...		XXXAnlEvent.TGo4EventE...			TFolder	
XXXAnlEvent.TGo4Ev...		XXXAnlEvent.TGo4EventE...			TFolder	
XXXAnlEvent.TGo4Ev...		XXXAnlEvent.TGo4EventE...			Bool_t	428
XXXAnlEvent.TGo4Ev...		XXXAnlEvent.TGo4EventE...			Short_t	428
XXXAnlEvent.frData[16]		XXXAnlEvent.frData[16]			Float_t	428
EventObjects		Event objects of curre...			TFolder	= 780
EventStores		References to event st...			TFolder	= 52
EventSources		References to event so...			TFolder	= 440

gui311

6.5.2 General functionality

Each item in the browser has a context menu, which can be activated by right mouse button click on that item. It is shown in the figure on the right. By means of this menu, it is possible to operate on the browsed objects. The items in the upper part of the context menu (above the line) are available for all items, whereas the items in the lower part contain special functionality to control remote data sources like the analysis.



gui311

Histograms and pictures can be plotted either by double click, or by drag and drop in a view panel, or by the right mouse menu. Item **Plot** draws each selected histogram into an own graphical pad, **Superimpose** draws all selected histograms superimposed on one pad.

The browser items represent the structure of a connected data source like the remote analysis, but will only retrieve the objects on demand. This happens usually just before the objects are drawn. To explicitely get the objects into the local memory cache without drawing them, the **Fetch item(s)** functionality may be used. Note that the browser's implicit fetching behaviour may be adjusted in the Settings/Preferences menu by "Fetch when drawing", and "Fetch when copying".

The selected objects may be saved into a ROOT file with menu item **Save selected...** The **Export to...** functionality will offer the possibility to export root histograms to ascii or radware format.

Item **Info** shows some information of the object, **Edit...** opens the editor if available. Item **Delete Item** deletes the selected objects from the local memory, whereas item **Delete from analysis** will delete the corresponding object in the remote analysis, if possible (see chapter 6.5.8. page 55).

6.5.3 Analysis folder controls

The **Analysis** folder shows the remote folder structure, which contains all objects that were registered to the analysis client. At any time the list of the remote objects may be refreshed by the right mouse button entry **Refresh nameslist**. The folder **Histograms** e.g. contains the histograms, the folder **Trees** will show the structure of all registered trees, e.g. all trees created by *TGo4FileStores*.
The eraser item **Clear (Reset to 0)** clears the selected objects like histograms, conditions, graphs and so on.
Each object on the analysis has two protection modes – delete protection and clear protection. These modes indicated in **Flags** column of analysis browser (see below). Delete protection is set for an object when it is created and added on the analysis side. It prevents deletion of such objects from GUI. Objects created by GUI commands have no such protection and can be deleted by the **Delete from analysis** functionality. Clear protection prevents the user to clear the content of objects by using . This mode can be set and unset for any object via context menu commands **Set clear protection** and **Unset clear protection** , respectively.

6.5.4 The monitoring mode

In the **Analysis** a histogram, graph, or picture can be set into the monitoring mode by selecting it and pressing the monitoring entry **Monitor item(s)** in the right mouse menu. This is indicated by the letter "**m**" in the **Flags** column of the browser (static objects have letter "**s**"). Monitoring means that the content of objects are updated continuously from the remote data source (analysis, histogram server,..) to the GUI. This allows e.g. to watch the filling process of a histogram. The monitoring property of an item may be switched off by the **Stop items monitoring** functionality of the context menu.
Note that only the visible objects are frequently updated, i.e. even if a browser object is in monitoring state, it will not be copied from the remote data source if is not drawn in any viewpanel, or displayed in an editor, respectively.
The overall monitoring action can be started with button of the **Browser options** dockwindow. Here the update frequency may be specified in seconds,too. Button will cease monitoring of all monitored objects, but will not change their monitoring property (flags). Additionally, this dockwindow offers a button for immediate refresh of all visible objects, and a filter function for the browser to display either all objects, or only the monitored objects, or only the currently fetched objects, respectively.

gui326

6.5.5 The workspace folder

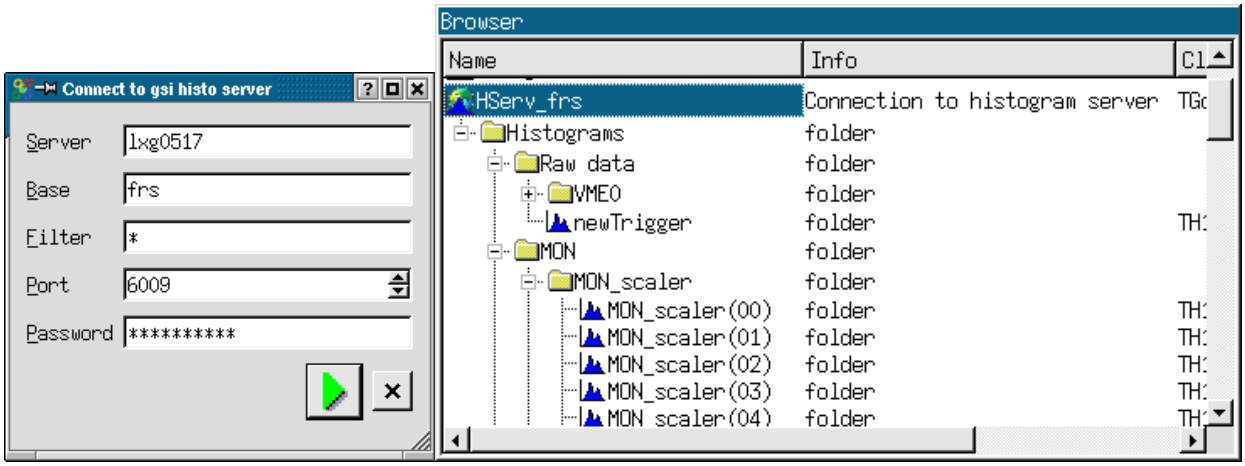
The **Workspace** folder contains all objects that are put into the memory of the local GUI. This may happen either by producing a new histogram from the ROOT menus in the viewpanel, like a rebinning, or a projection, or from the Go4 tree viewer; or objects may be copied from elsewhere to the workspace. Item **Copy to Workspace** will produce a copy of the current object and put it into the workspace folder. This copy will preserve the subfolder structure of the data source; if e.g. a histogram was copied from analysis folder "Histograms/Crate1", the copy will be placed in folder "Workspace/Analysis/Histograms/Crate1". The **Copy to clipboard**, **Paste from clipboard**, respectively, allow a standard copy/paste functionality to any destination in the workspace. Additionally, in the workspace folder the right mouse button menu offers the **Create folder** and the **Rename object** functionality, as known from general file system browsers.

6.5.6 Browsing files

ROOT files containing data can be opened (buttons and of the mainwindow file menu, respectively) as with the native ROOT *TBrowser/TTreeViewer*. Any ROOT file can be opened. Histograms in these files can be displayed in the Go4 view panel like local objects. A ROOT tree in a local file can be examined with the tree viewer of Go4. In contrast to the remote tree viewer mode, trees in a local file are processed by the GUI itself and do not have an effect on the remote analysis. The GUI knows if a tree viewer entry comes from a remote, or from a local TTree, so the button will either send a command to the analysis client for a dynamic histogram, or will perform a local *TTree::Draw()* call.
If the file contains user objects, make sure that the GUI has loaded the proper libraries to access them (see chapter 6.2, page 45).

6.5.7 Histogram server connection

From the main window File menu entry one can connect to any GSI histogram server like MBS, GOOSY, LeA, or another Go4 analysis. The parameters for the histogram server, such as node name (**Server**), login name (**Base**), the socket **Port** number, the **Password**, and an optional **Filter** expression, are specified in a connection dialogue window. After a successful connection the histograms of the server appear in the Go4 browser in a folder named **HServ**_***basename***, if basename is the name of the histogram server base.



gui316/317

6.5.8 Resetting and deleting objects

Any object in the workspace may be deleted by selecting it and using the popup menu delete item . Objects in the Analysis (histograms, conditions, parameters, …) that were created in analysis code must not be deleted, for the compiled user analysis would still try to access these objects after deletion. Therefore, deleting these objects is disabled using the delete protection property (symbol "p" in **Flags** browser). However, dynamic objects that had been created from the gui (histograms, conditions, dynamic list connections) are not delete protected and can be removed by the delete button.

An analysis histogram can be reset (contents and statistic values to zero) by selecting it and pressing the clear button except Clear is disabled. Resetting an analysis *TGraph* object will erase all points of the curve. For parameters, the method *Clear()* is called which may be implemented by the user. All objects within an analysis folder are reset at once by selecting the folder icon in the remote browser and pressing button . This has the same effect as calling method *ClearObjects("Foldername")* of *TGo4Analysis*. Note that any analysis object can be protected against clearing by a switch in the remote browser's right mouse button context menu (See chapter 6.5.3).

6.6 *The Go4 tree viewer*

The Go4 tree viewer is started via **Settings►Show/Hide►Tree viewer** menu or via RMB pull down menu.

gui140

There are two operation modes for the Go4 tree viewer: the local mode, or the remote mode. Dragging and dropping the tree leaf names from file or remote browser, the tree viewer will switch automatically into the local or remote mode, respectively.

6.6.1 Local mode

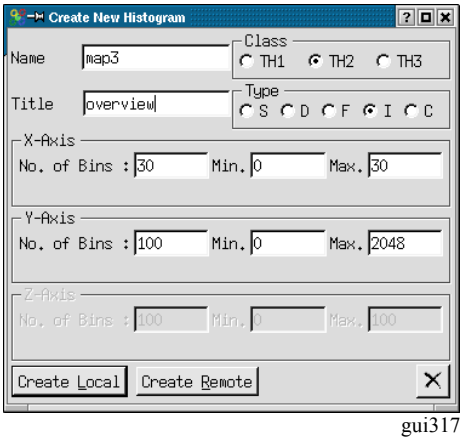
The tree viewer works on a tree in a file that was opened in the browser. This is like the original ROOT tree viewer, with the same logic of drag and drop. However, the Go4 tree viewer supports the resolution of the Go4 composite event information (see section 7, page 80). On pressing button , the local tree will be processed as defined by the given draw expressions in X: Y: Z: (and optional) fields of the Go4 tree viewer. The local histogram of the given name is filled with the result. The histogram will appear in the memory tab and may be displayed in a view panel. If no name is specified, an automatic name is chosen from the given leaf names.
All classes, which are stored in the tree, should be known to GUI. User should load appropriate libraries before using local tree viewer (see chapter 6.2, page 45).

6.6.2 Remote mode (dynamic list histogram)

The **Analysis** folder shows the structure of all objects registered to analysis trees in the Trees subfolder. By drag and drop the elements of a tree can be put into X: Y: Z: fields of the Go4 tree viewer. A name and an optional drawing condition can also be defined here. The logic is the same as for the regular ROOT tree viewer. On pressing button , this information is passed to the analysis client and a new entry in the Go4 dynamic list is created. After pressing in the Analysis panel, a new histogram of the defined name appears in the histogram folder (if no name was defined in the tree viewer, a default name is used combining the variable names). Note: the histogram itself will be created no sooner than the next events after the are processed, i.e. the analysis must be running. This histogram will be filled event by event with the defined parameters of the tree. Go4 internally uses a *TTree::Draw()* over a number of collected events to update the histogram contents. This number, the dynamic list interval **TreeDrawInterval**, can be set by the analysis method *SetDynListInterval(Ndyn)*, or can be changed in the dynamic list editor (see chapter 6.12, page 74).
If the histogram specified in the tree viewer already exists when the dynamic list entry is created, the histogram of that name will be filled by the dynamic list instead of filling a new histogram. Therefore it is possible to create a histogram with desired bin size first (see chapter 6.6.3, page 56), and then assign this histogram to a new entry of the dynamic list. This can be done easily by dragging and dropping a histogram icon from the histograms folder into the histogram text-box of the tree viewer. Again, pressing will create the dynamic list entry; the given histogram will then be filled every *Ndyn* events. The dynamic list tree is kept in memory, if in the analysis configuration for output **Go4BackStore** had been selected.
A histogram filled by the dynamic list, like any other remote histogram, can be displayed continuously in a view panel by switching on the Go4 monitoring mode (see chapter 6.5.4, page 54).

6.6.3 Creating a new histogram

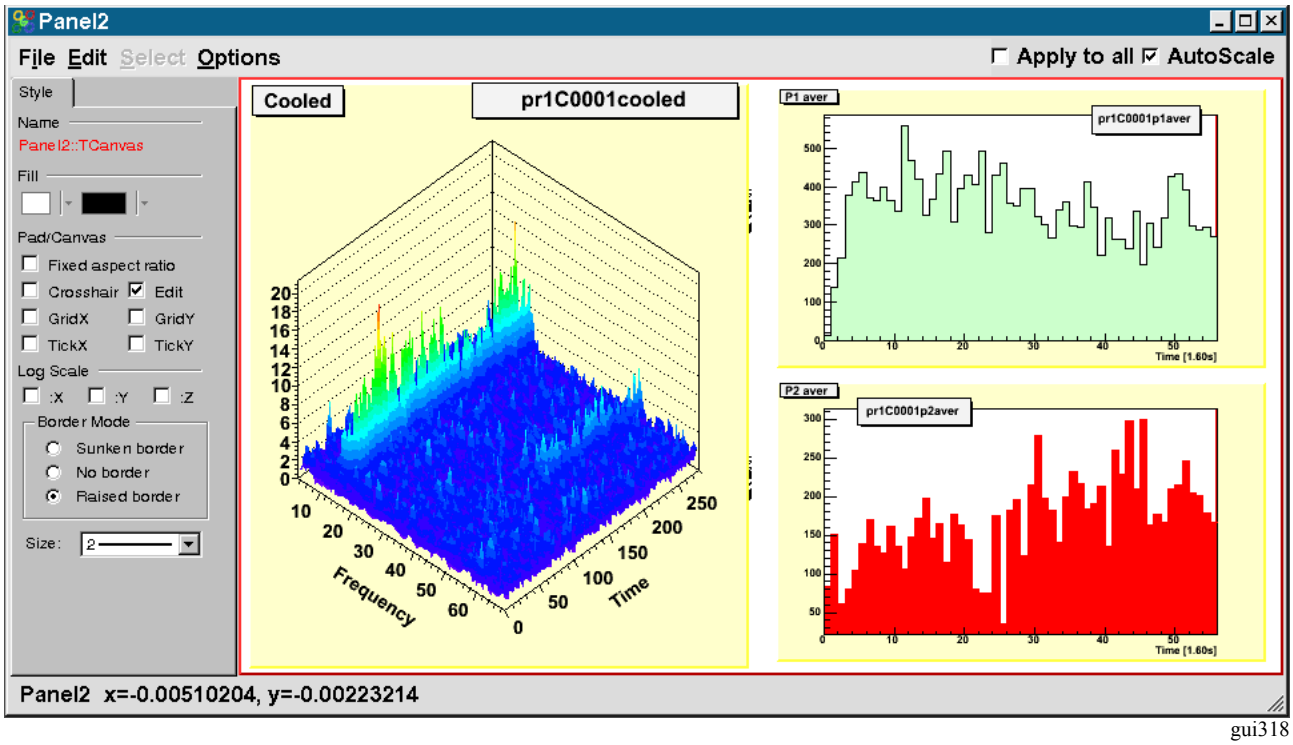
The button will popup the histogram creation window. Here the properties of the histogram to be created anew can be specified (dimensions, precision, binning, range, name, title). The histogram may be either created in the local directory (**Create Local**), or created in the remote analysis (**Create Remote**). A new local histogram will appear in the local objects panel, a remote histogram is put under the histograms folder in the Go4 folder structure. A new histogram (like any existing histogram) can be used as target for the remote or local tree viewer. This is done by specifying the histogram name in the tree viewer name field, or by dragging and dropping the histogram icon to this name field. The tree viewer will then fill the created histogram instead of creating a new histogram with arbitrary binning and range settings.



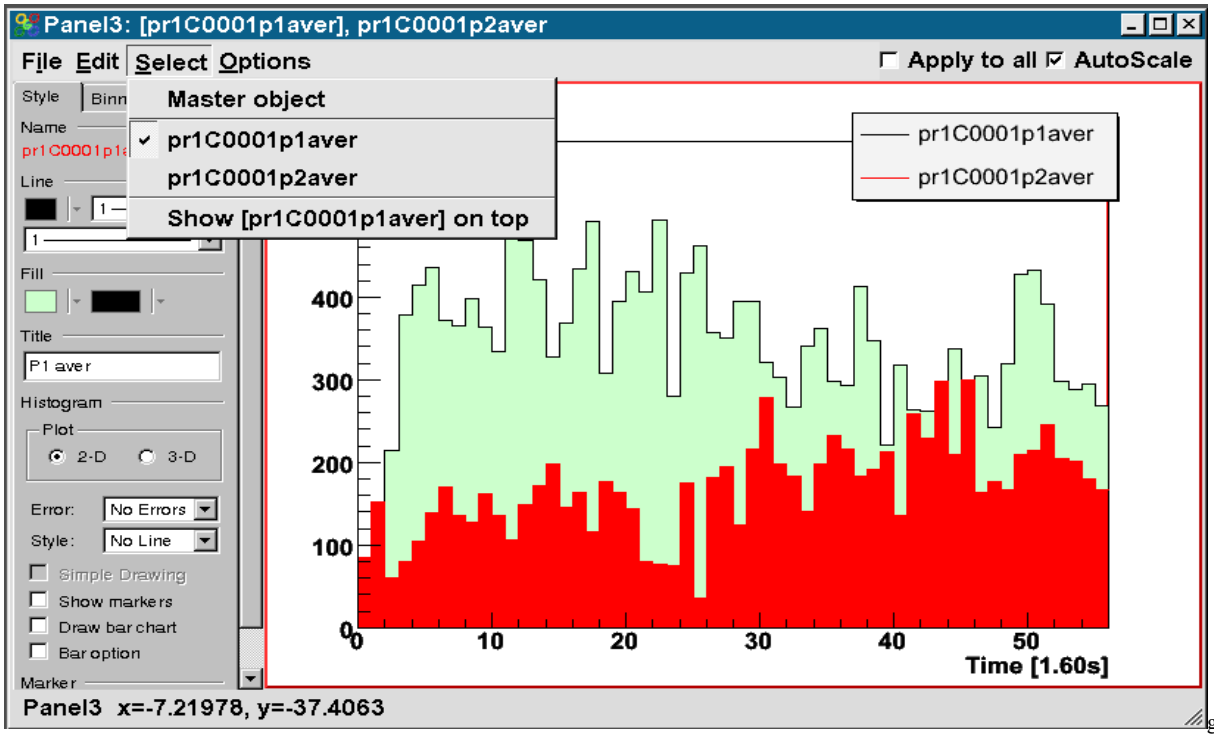
gui317

6.7 *The Go4 view-panel*

Pressing in the Go4 main control window opens a new Go4 view panel. A new view panel will also pop up automatically when any object in the browser is selected and the right mouse button menus or are activated. Furthermore, objects can be drawn by "drag and drop" from the Go4 Browser to an existing view panel pad and displayed there. On the left side the optional ROOT graphical editor is embedded. It is opened by **Edit►Show ROOT attributes editor**. Select with left mouse an object on the canvas and the editor will change accordingly.



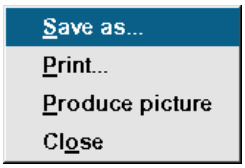
gui318



gui368

The view-panel offers the menus:

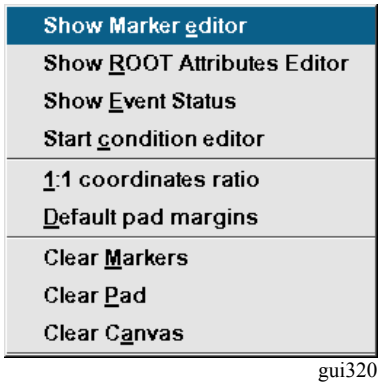
6.7.1 File menu



gui319

Save as..	save the content of the view-panel in different formats.
Print ...	hardcopy the view-panel to $PRINTER or .ps file
Produce Picture	create Go4 picture from viewpanel, put it in workspace
Close	the view-panel

6.7.2 Edit menu

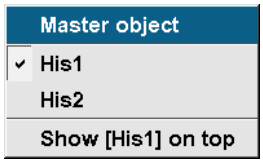


gui320

Show Marker Editor	open marker panel
Show ROOT Attributes	open ROOT graphics editor
Show Event Status	toggle ROOT event status in bottom line
1:1 coordinate ratio	adjust pad margins to 1:1 coordinate ratio
Default pad margins	restore default pad margins
Clear Markers	clear all marker objects in pad
Clear Pad	clear contents of current pad (and sub-pads)
Clear Canvas	removes content and pad divisions

6.7.3 Select menu

When histograms or graphs are displayed in superimpose mode, each one may be selected here. Then attributes like color may be set for selected histogram. If the selected object is currently not on front of all superimposed objects, an additional menu entry "**Show ... on top"** will appear. When chosen, this entry will pop the selected object to the foreground. Note that the object first must be selected and then set to top.



gui320a

6.7.4 Options menu

gui159

Crosshair	toggle the ROOT pad crosshair mode
Super Impose	toggle superimpose option
Histogram Statistics	toggle display statistics box on pad
Multiplot Legend	show legend for superimposed histograms
Histogram Title	toggle display histogram title on pad
Draw Time	display refresh time in histogram title box
Draw Date	display refresh date in histogram title box
Draw item name	display full path and name in histogram title box
Keep View panel Title	Do not overwrite title
Set View panel Title	Set the title

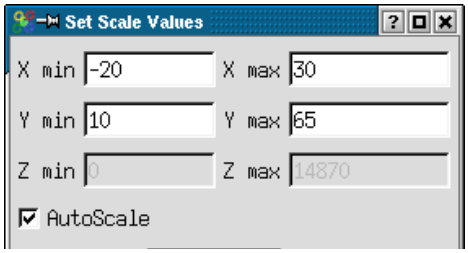
With **Settings->Panel defaults** one can set defaults for these values. If the **Superimpose** option is selected, any new histogram that is dragged to this pad will not replace the existing histogram, but will be displayed in the same pad with the old one (as ROOT *THStack*). A legend box will show the graphical style and the name for each drawn curve. This legend can be toggled on or off with the **Multiplot Legend** option. The text of each legend entry can be changed by opening the right mouse button popup menu at the entry position and using the **SetEntryLabel** function (see ROOT TLegend class for documentation of further methods in this menu).
An existing view panel can be divided into independent sub-pads by the division buttons in the **Canvas Tools** activated with the RMB on an empty region. When several histograms in the browser are selected for plotting, the view panel division will be done automatically to display all histograms in one new view panel window. Graphic style and range settings are always applied to the sub-pad that was selected most recently (red frame which is set by middle mouse button in ROOT), except the **Apply to all** option checkbox is enabled.
It is possible to extend the regular histogram title by information on the refresh time and date by switching on the options **Draw Time ,** and **Draw Date** ,respectively. Additionally, the full name of the displayed object, i.e. the complete path and item name in the Go4 browser, may be displayed in the histogram title by toggling the **Draw item name** option.

Usually, the title of the view panel window (showing up in the **Windows** menu of the main Go4 window) is taken from the object that was drawn most recently in one of the sub-pads. This behavior can be changed by options **Keep View panel Title** and **Set View panel Title** , respectively. This allows to specify a meaningful name for a view of several histograms that will not change when one histogram is exchanged by drag and drop on a sub-pad.

The **Show event Status** option in the edit menu will display **the current mouse coordinates and histogram channel contents** in the bottom line of the view panel. If the canvas is divided, this information always refers to the selected pad.
The canvas embedded in the Go4 View panel is an ordinary ROOT canvas, offering all ROOT features of the mouse button actions on the displayed objects (e.g. opening a histogram fit panel, rescaling the axes using cursor and left mouse button, ...). The active pad must be selected with middle mouse (ROOT). **After using ROOT popup windows the active pad must be selected again!** Note that the settings are preserved for each pad!

The view panel may be saved to a file in several formats by choosing **File►Save as**.

The buttons  (gui141) are zoom and shift buttons for the x-, y and z- axes, working on the active pad. In multi pad view panels the active pad must be selected with middle mouse (ROOT, red frame). **After using ROOT popup windows the active pad must be selected again!** The expansion/compression factor can be set in % of the current range. The **Un-zoom all** button will restore the complete range of all axes. The **set limits** button will popup a scale window. Here the range can be typed in and set explicitly by axis values. Additionally, the scaling behavior of the ROOT histogram can be changed: By default (**AutoScale** on), the y-axis (1D histogram) or z-axis (2D histograms), respectively, is expanded to cover the full range of channel contents whenever a memory histogram is updated, or when a monitored histogram is refreshed from the analysis. With **AutoScale** disabled, the previous y-range (1D) or z-range (2D), respectively, is invariant over any updates. This allows to observe a magnified region of interest in a spectrum, independent of the maximum peak height. Note that the y range of a 1D histogram can be chosen freely by ROOT TAxis selection with the mouse, i.e. clicking with left mouse button on the y-axis for the first limit, and dragging the pressed mouse to the second limit of the range. The scale window is automatically connected to the selected pad and updated accordingly.



6.7.5 List of draw options

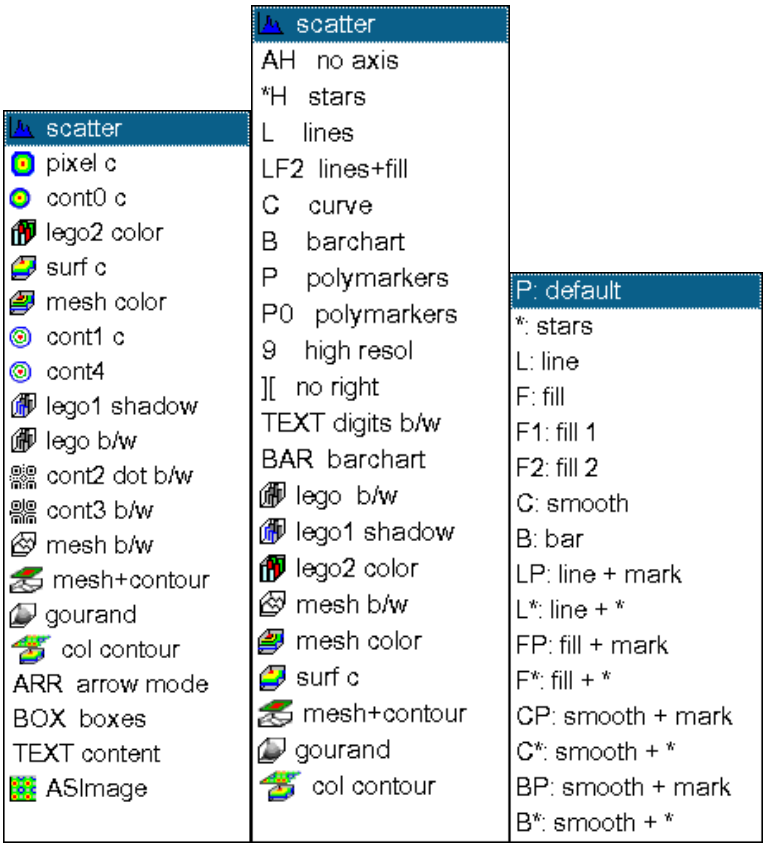
The draw options can be set by two menu bars: One for all options available (**Settings->Show/hide->Draw Options**) steered by pull down menus and one for a subset (**Settings->Show/hide->Draw Options Short**) steered by buttons only:

gui366

The draw options follow the ROOT draw options (see next page).

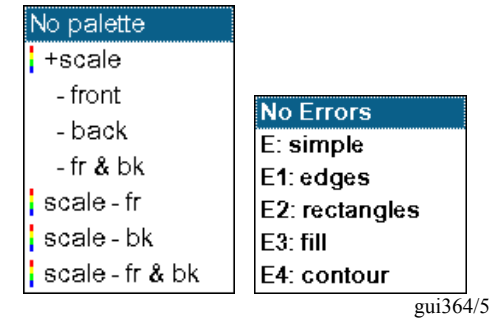

gui367

Draw options for 2-dim and 1-dim histograms and graphs:



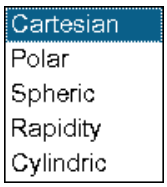
gui135/gui362/9

Details for 2-dim and 1-dim histograms:



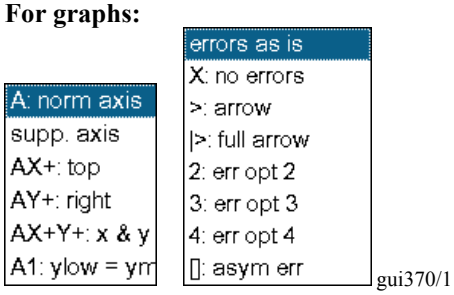
gui364/5

Coordinate system:



gui365

For graphs:

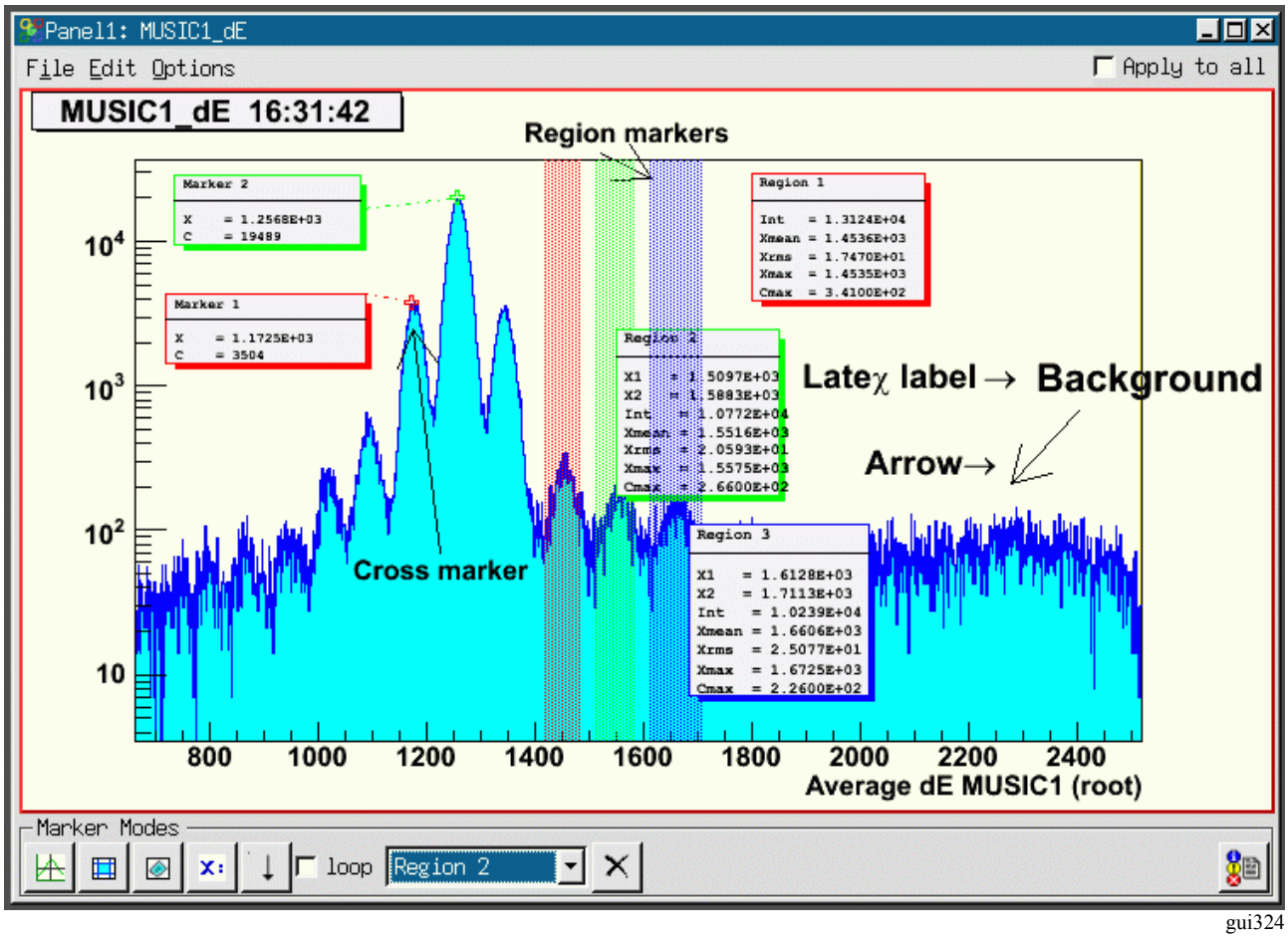


gui370/1

Go4 option	Description	ROOT
scatter	black scattered points	HIST
pixel c	colored pixels	COL
cont c	colored contour	CONT
surf c	colored surface	SURF2
pix+scale c	colored pixels and color scale bar	COLZ
cont+scale c	colored contour and color scale bar	CONTZ
Gouraud	smooth grey scale surface	SURF4
lego c	colored lego	LEGO2
lego/shadow	lego with one side colored	LEGO1
lego bw	black and white lego	LEGO3
mesh c	colored meshed surface	SURF1
mesh bw	black and white meshed surface	SURF
mesh+cont	bw meshed surface and colored contour on top	SURF3
line c	colored contour lines	CONT1
line dot bw	black dotted contour lines	CONT2
line bw	black contour lines	CONT3
boxes bw	black boxes	BOX
digits bw	channel content as numbers	TEXT
ASImage	TH2 as TASImage (fast pixel map with scale bar)	
P0 (1D)	Polymarker without lines	P0
L (1D)	Line	L
C (1D)	Smooth curve	C
B (1D)	Bar chart	B
mesh+cont2	bw meshed surface and colored contour on top	SURF5
cont4	colored contour	CONT4
cont1+ pal	colored contour lines and color scale bar	CONT1
cont4+pal	colored contour and color scale bar	CONT4
arr (2d)	arrow plot	ARR

6.7.6 Channel and window markers

In a view panel a marker panel can be opened by **Edit►Show Marker Editor** menu item:



gui324

Pressing once on button and then one more time in the pad, a channel marker (cross) with a label and a connecting line is drawn. Once created, any marker can be re-positioned by chosing its name in the marker selection box and using again the button: the next pad click moves the currently active marker to the picked position. If **new** is chosen in the marker selection box, a new marker is created and added to the list. Note that the selected marker is always displayed on front of all other objects in the pad. Clicking on a marker or its label box with the left mouse button will also pop it frontmost.

With **new** selected and **loop** option enabled, the cursor stays after in point marker mode. Subsequent clicks in the pad create new markers. This behavior also applies for the other marker types, respectively:

draws a window marker (with two subsequent LMB clicks) and a label.

draws a polygon marker (*TCutG*): each click will define one point of the polygon, a double click will finish the definition of the shape.

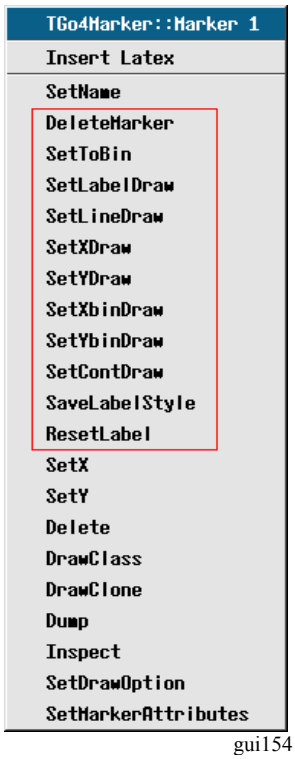
X: places a (Latex formatted) label.

draws an arrow from first click to second click.

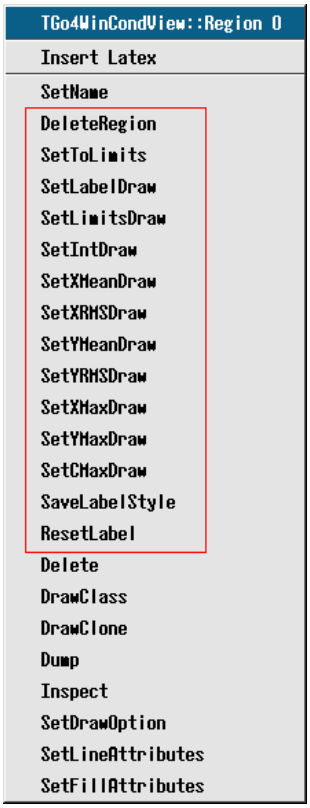
In **loop** mode one can switch between the five marker types.

outputs the values of the markers to the activated log output.

A selected markers can be deleted by pressing the button near the marker selection box. Furthermore, markers may be deleted and configured with RMB on the cross or inside the window, respectively (see right **TGo4Marker** menu: **DeleteMarker** and

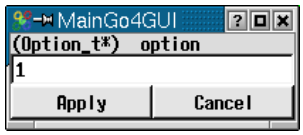


gui154



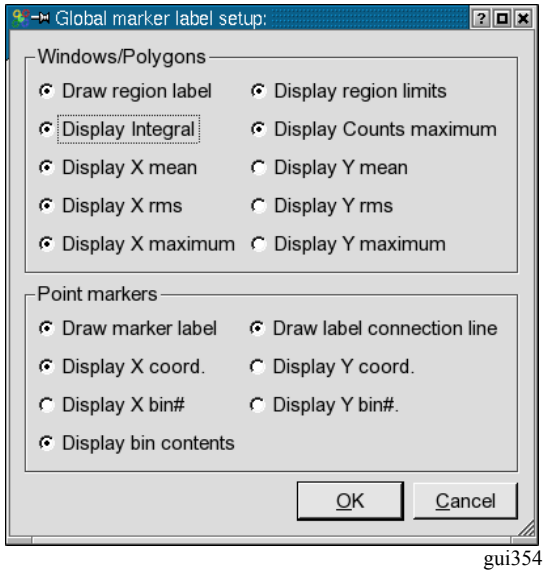
gui155

left **TGo4WinCondView** menu: **DeleteRegion**). The setter methods configure the layout through little windows as shown above (options 0 or 1, then apply and cancel). All elements can be moved with LMB (labels are updated). **SaveLabelStyle** applies current settings to all subsequent markers. With **Settings►Save settings** in the main Go4 window menu these settings will be stored. With **Edit►Clear Markers** one can remove all marker elements. To change the graphical attributes one can use the new ROOT graphical editor. It should be opened by **Edit►Show ROOT attributes editor.** When a graphical object is selected (LMB) the editor changes accordingly. Close the editors also through the **Edit** menu.



gui156

With **Settings►Panel defaults►Marker labels** one gets the window shown below. Here the default layout can be specified and saved.

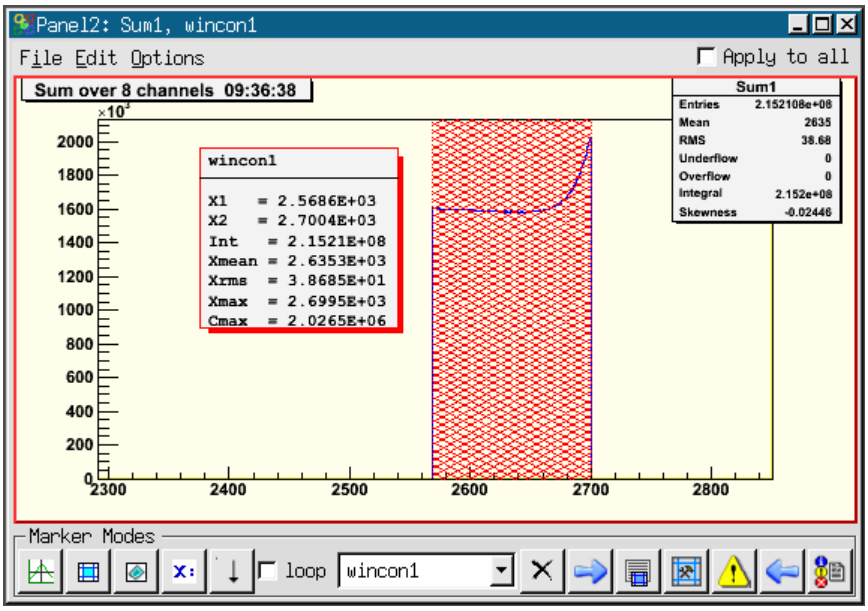


gui354

6.8 *Conditions*

6.8.1 Conditions editing in viewpanel marker editor

A condition may be displayed in an existing viewpanel by dragging and dropping it from the browser to a destination pad containing a appropriate histogram. The full condition editor (see 6.8.2) may also draw its working condition to the viewpanel.
It is possible to edit any condition displayed in a viewpanel already by means of the marker editor in the bottom line (see figure).

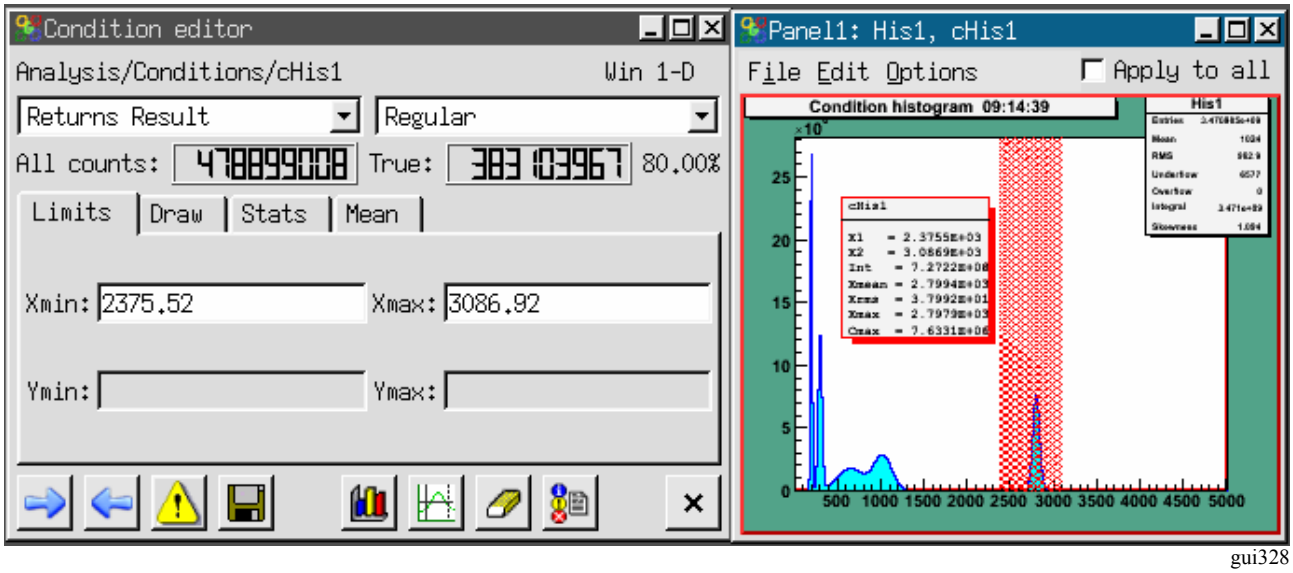


gui330

Condition wincon 1 is drawn above the histogram Sum1 that is filled only if this condition is true. As the regular markers (see 6.7.6), the condition may be selected by name in the marker selection box. In addition to the control buttons for the markers, editing a condition will enable some more buttons in the marker editor. After changing the condition by moving its boundaries, a ⚠ will appear to remind you to update the condition by button ⬅ on the analysis side. With ➡ the current condition state from the analysis side is refreshed in the editor window. If working on a condition from file, the refresh button will appear instead to reload the viewpanel condition.
Button opens the info window for the selected condition (see 6.13) to view current conditon properties that are not displayed in the viewpanel label. For advanced editing of the condition, the full condition editor may be invoked using button .

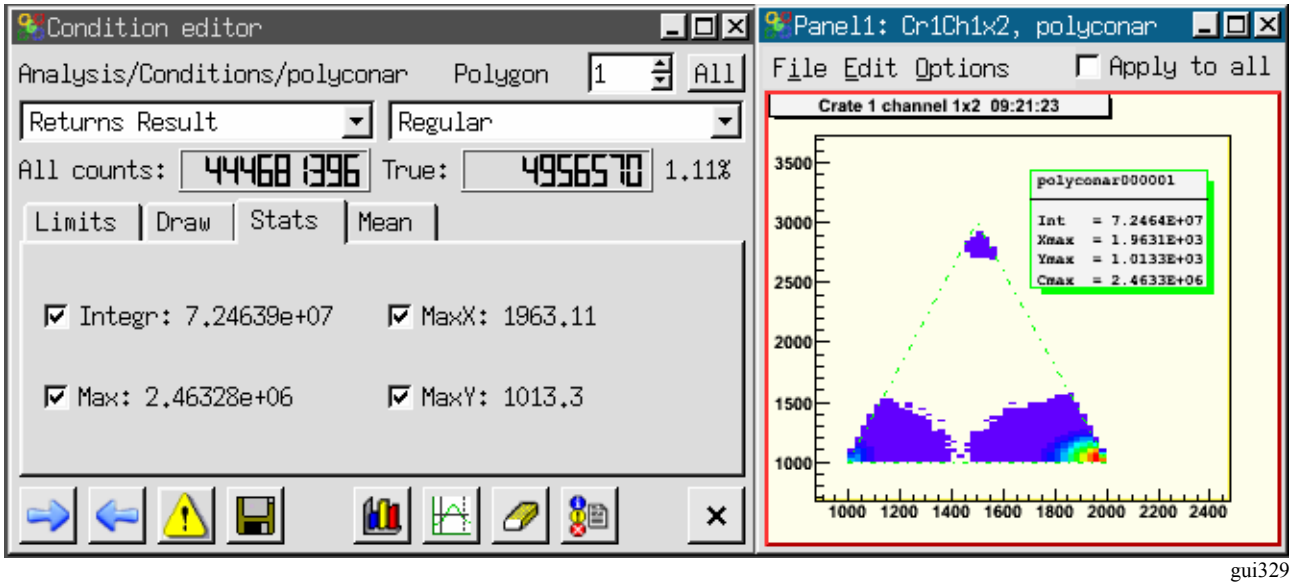
6.8.2 Full condition editor

The condition editor window is popped up when one double clicks on a condition in the browser or using the edit function of the browser's right mouse menu. It may also open by using the in the viewpanel marker editor.
In addition to the features of the marker editor, it may display and change all properties of the Go4 condition class, e.g. counters, testing properties, histogram statistics over the region, etc.
The following figures give some examples.



gui328

Window condition `cHis1` displayed with histogram `His1`. The histogram has been bound to the condition by method *SetHistogram()* in the analysis. In this case the histogram is automatically displayed when the condition is edited.
Polygon condition `polyconar` is a polygon condition array from the two step example which can be displayed in a 2d view panel. When a **condition array** compound is edited, the index of the currently active condition can be set in the upper right spin box. The displayed values always refer to the selected array member. When selecting an entire condition array in the editor (**All** button or spin box index "-1"), changes will be applied to all members.



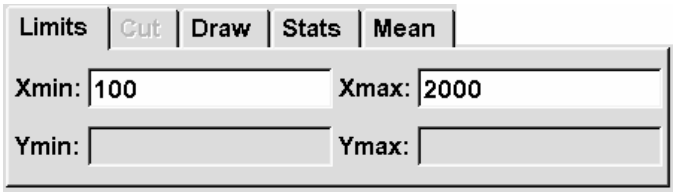
gui329

With the button the active pad of the current view-panel (selected with middle mouse button) is set as display working pad for the condition. The condition is drawn on this pad until the display button is pressed again with another active pad. If the working pad contains a histogram, it is assigned to the condition under edit and its name is shown in the editor. Note that it is possible to exchange the condition work histogram by drag and drop of a new histogram into the condition editor display pad.
After editing the condition limits graphically on the working pad, the changes will be updated automatically whenever the mouse enters the editor window. When a condition is changed in the editor (always press Enter to confirm changes), the graphical representation will be updated automatically. After changing the condition, a will appear to remind you to update the condition by on the analysis side. With the current values (e.g. counters) from the analysis side are

updated in the editor window. Conditions can be set to return always true or false, respectively. The result of a condition check can be inverted. A polygon condition checks, if a point (x,y) is inside a polygon (*TCutG*). A window condition checks, if one or two values are inside one or two intervals, respectively.
A condition has counters for the number of all *Test()* calls performed, and for the number of true results. The counter values after the last refresh are displayed in the editor. With these values are reset to zero and **the condition is directly updated on the analysis side**.
The  button allows to pick the boundaries of the condition region with the mouse. This works in the same way as in the marker editor: for window condtions, two subsequent clicks will take the click position as limits (for 2d conditions, these clicks define corner points); for polygon conditons, each click will set a corner point until the mouse double click finishes the pick mode.
Button  outputs the current condition values to the GUI starting window, or into a log file if specified in the Settings menu (see 6.1). Button  saves the condition in a file. If the condition editor is working on a condition in a ROOT file (via File Browser), the  button will update the changes in the original file by default. This is useful to edit conditions in an existing auto save file.

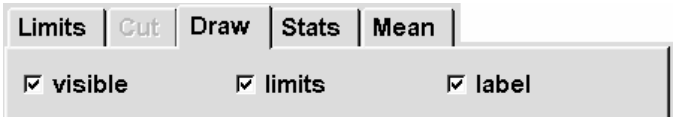
6.8.3 Editor tabs

The condition editor offers four tabs: for the condition limits, for the display properties, for the statistics inside the selected condition range, and for the mean values, respectivly They are shown in the next screen shots:
The **Limits** tab contains the values of the window condition limits, or the largest extension of the polygon condition boundaries. These are updated from the graphical representation on the working pad, or can be typed in directly in case of window conditions (to apply the typed values press RETURN).

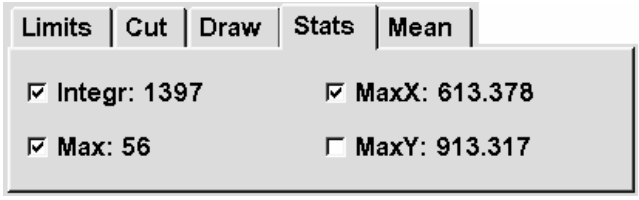


gui331

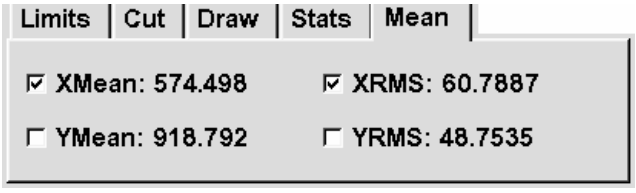
The **Cut** tab is only active for polygon conditions. It shows the table of x and y coordinates of the polygon (*TCutG).* These values may be edited here (to apply the typed values press RETURN). Moreover, the number of polygon points can be changed with the **NPoints** selector box. If the *TCutG* is edited graphically on the pad by mouse, the values in the table will be synchronized the next time the mouse enters the editor window.
The **Draw** tab shows the names of the histogram and viewpad used to display the edited condition, and allows to control some draw properties. Each condition can be set as visible or not with the **visible** checkbox. If visible, the condition is shown on the working pad, otherwise it is hidden. This is useful when working with condition arrays. It is recommended for polygon conditions to improve editing. The visibility is a property of the condition class itself and is stored in the auto-save file. The **label** checkbox enables the drawing of a graphical label together with the condition (see screenshot examples). This label may contain the limits values from the Limits tab; this can be toggled using the **limits** checkbox. Other entries of the label may be configured in the **Stats** and **Mean** tabs.



gui332

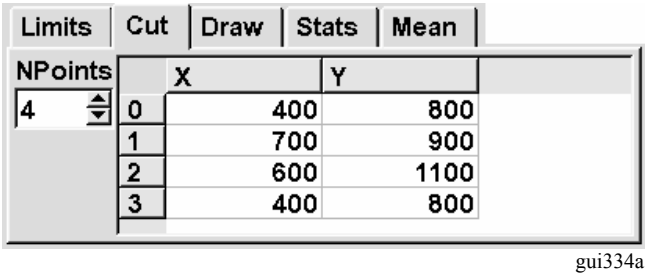


gui333



gui334

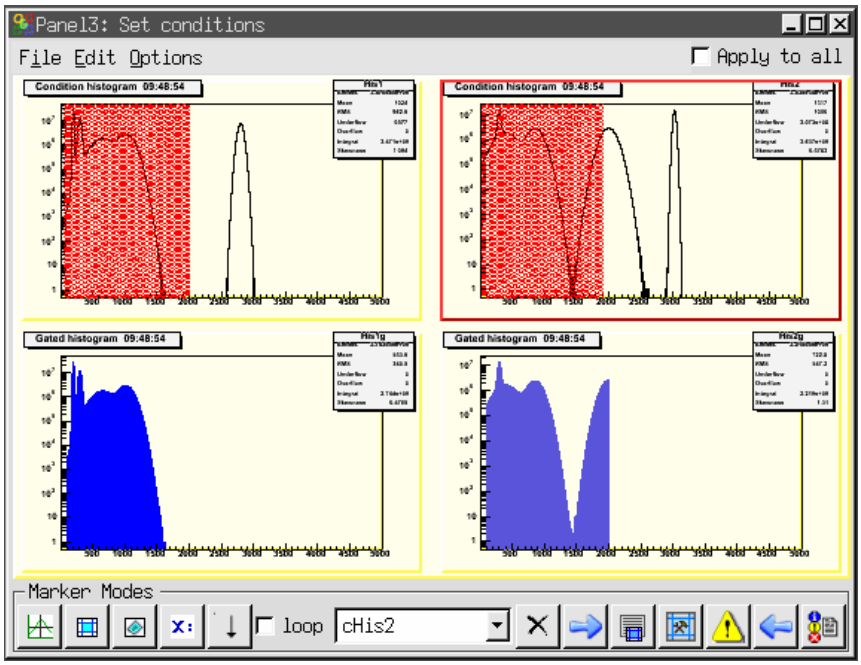
The **Stats** tab shows some statistics (Integral, position and channel content of the maximum) of the current histogram inside the selected condition. In addition, the **Mean** tab contains mean and RMS values for x and y directions. Setting the corresponding checkboxes plots these values into the label on the working pad.



gui334a

6.8.4 Conditions bound to pictures

In the next example two conditions are bound to the upper pads of a picture (see chapter 6.9, page 70) by method *AddCondition()*.

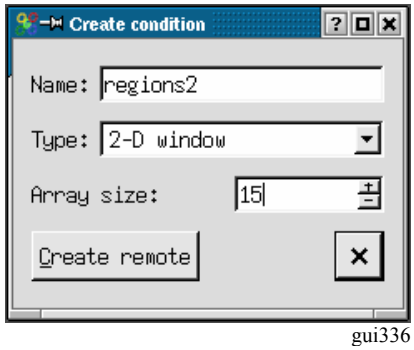


gui335

The histograms in the lower pads are filled under the condition shown in the pad above. All picture conditions will be shown simultaneuosly (if their visible property is true). Mouse click on a picture's subpad will deliver the names of all contained conditions into the selection box of the marker editor. The selected condition may be modified and updated by means of the marker editor, or using the full editor started by button, as described above. The mechanism to bind conditions to picture pads guarantees that a condition is set always on the correct histogram.

6.8.5 Creating conditions

With the button of the main window "Tools" menu and toolbar, one can open a window to create a new condition in the analysis. This functionality is available as a shortcut from the dynamic list editor, too (see 6.12). The Create condition dialogue expects a condition name, the type (1-D window, 2-D window, polygon), and optional an array size. For Array size "no array", a single condition is created, otherwise a condition array compound that contains the given number of conditions.



gui336

After pressing the Create remote button, the new condition will appear in the subfolder Analysis/Conditions of the Go4 browser. The name field in the create dialog may contain any subfolder path relative to this default location, e.g. `Name: myconditions/region2` will create new condition region2 in folder Analysis/Conditions/myconditions. Non exisiting subfolders are created in this procedure together with the conditon.

Once created, the condition can be modified from the condition editor or from the viewpanel marker editor as described above. When the auto-save mechanism was enabled, the condition will be restored at next analysis startup. Note that it's not possible to create a new condition without the analysis connected to the gui!

6.9 *Pictures*

The *TGo4Picture* class provides a way to set up a view in the analysis, which then can be displayed in the Go4 GUI. A picture contains:

- references to objects (via names), which should be displayed;

- division setups of pictures into sub-pictures;
- draw options and parameters like line attributes, axis ranges and so on.

The following code creates a simple picture, which contains only one histogram:

```
TGo4Picture* pic = new TGo4Picture("pic1","picture title");
pic->AddH1(histo);  // histo is variable of type TH1*
```

A picture can be divided into sub-pictures like a ROOT canvas can be divided into sub-pads. The division of a picture can be specified in the picture constructor or by method *SetDivision(int ndivy, int ndivx)* which creates ndixy*ndivx sub-pictures inside the picture. Sub-pictures can be accessed via method *Pic(posx, posy)*. For each picture (and sub-picture) one can specify the following options:

Display header	`pic->SetDrawHeader()`
X axis range	`pic->SetRangeX(double, double)`
Y axis range	`pic->SetRangeY(double, double)`
X log scale	`pic->SetLogScale(0, bool)`
Y log scale	`pic->SetLogScale(1, bool)`
Z log scale	`pic->SetLogScale(2, bool)`

To add an object to be drawn the following methods can be used:

TH1, TH2, TH3	`pic->AddH1(TH1*)`
THStack	`pic->AddHStack(THStack*)`
TGraph	`pic->AddGraph(TGraph*)`
TGo4Condition	`pic->AddCondition(TGo4Condition*)`

Each method requires a pointer to the correspondent object and optional draw options (if necessary). When an object has been added to a picture, the following drawing options can be set for this object (see ROOT manuals):

Line attributes	`pic->SetLineAtt(Color_t, Style_t, Width_t)`
Fill attributes	`pic->SetFillAtt(Color_t, Style_t)`
Marker attributes	`pic->SetMarkerAtt(Color_t, Size_t, Style_t)`
Draw options	`pic->SetDrawOption(Option_t *)`
TStyle attributes	`pic->SetStyle(TStyle*)`
Axis rebining	`pic->SetRebinX(Int_t ngroupx), pic->SetRebinY(Int_t ngroupy)`

For example, to configure a picture with four sub-pads (2 x 2), each with a different histogram, the following code can be used (**first index top down, second left right**):

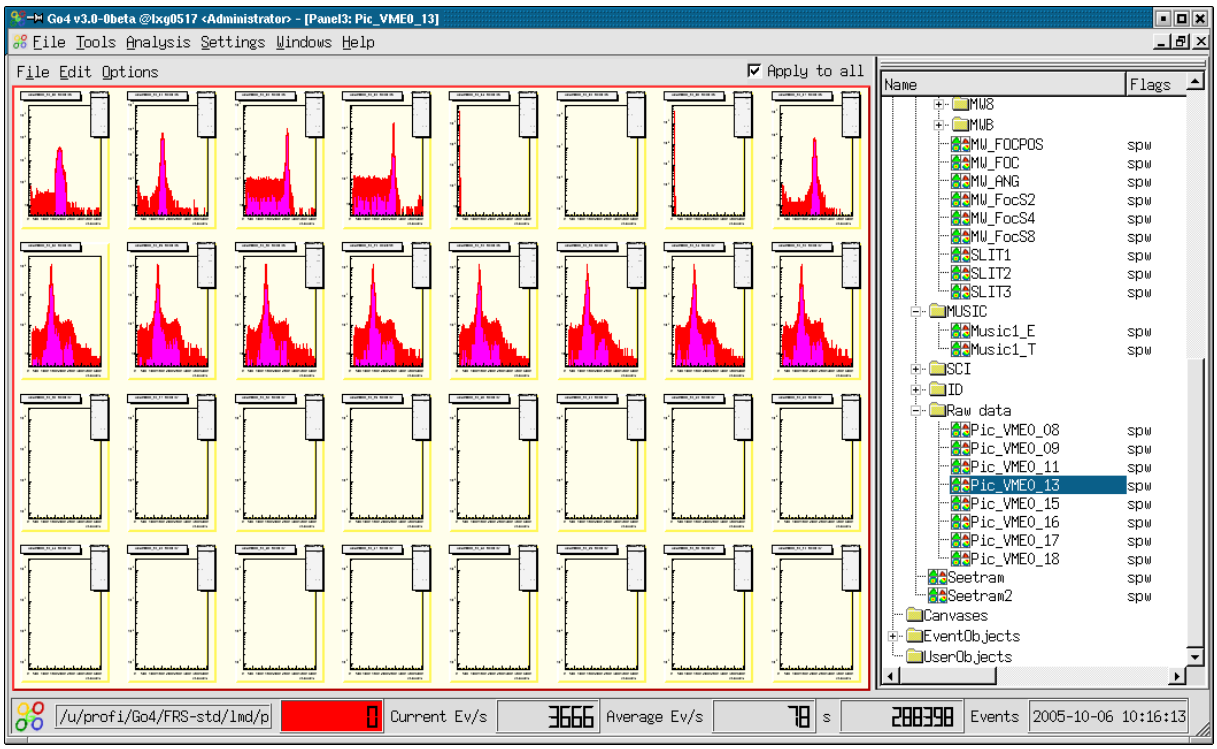
```
TGo4Picture* pic = new TGo4Picture("pic1", "picture title", 2, 2);
pic->SetDrawHeader(kTRUE); // displays time, name and title of picture
pic->Pic(0,0)->AddH1(histo1);
pic->Pic(0,0)->SetRangeX(100, 200);
pic->AddH1(0, 1, histo2);  // or pic->Pic(0,1)->AddH1(histo2);
pic->Pic(0,1)->SetDrawOption("lego");
pic->AddH1(1, 0, histo3, "lego");
pic->AddH1(1, 1, histo4);
AddPicture(pic); // add picture to frame work
```

Similarly the colors in next figure have been set up by:

```
Color_t his=0;
for(int i=0;i<8;i++) for(int k=0;k<8;k++){
   fPict1->Pic(i,k)->SetFillAtt(his,1001);
   fPict1->Pic(i,k)->SetLineAtt(his,1,1);
   his+=2;
}
```

The *TGo4Picture* class supports arbitrary levels of picture divisions. This means that each sub-picture can also be divided. For instance, a picture with 3 histograms, two in top row and third in bottom row, will be created by the following code:

```
TGo4Picture* pic = new TGo4Picture("pic","pic title",2,1);
pic->SetDrawHeader();
pic->Pic(0,0)->SetDivision(1,2);         // divide top widget on two more pads
pic->Pic(0,0)->Pic(0,0)->AddH1(histo1);  // add histogram to sub-sub-pad
pic->Pic(0,0)->Pic(0,1)->AddH1(histo2);  // add histogram to sub-sub-pad
pic->Pic(1,0)->AddH1(histo1, "lego2");   // add histogram to sub-pad
AddPicture(pic);
```



gui337

Current limitations of pictures are:

- Only histograms *(TH1)*, graphs *(TGraph)* and stacks *(THStack)* can be add to picture or sub-picture.
- Several histograms or graphs displayed together only when pic->SetSuperimpose(true) is set.
- Conditions can be displayed only in pair with a histogram.
- A condition can be added only after a histogram has been added.

In the Go4 GUI pictures will appear in the analysis browser in the **Pictures** subfolder. Together with the picture all correspondent histograms will be automatically transferred. Double click on a picture draws it in a new view panel. A picture also can directly drag-and-dropped into an existing view panel.
Pictures also can be put to the monitoring list. Putting a picture to the monitoring list automatically puts all histograms of the picture to the monitoring list, too.

6.10 *Fit GUI*

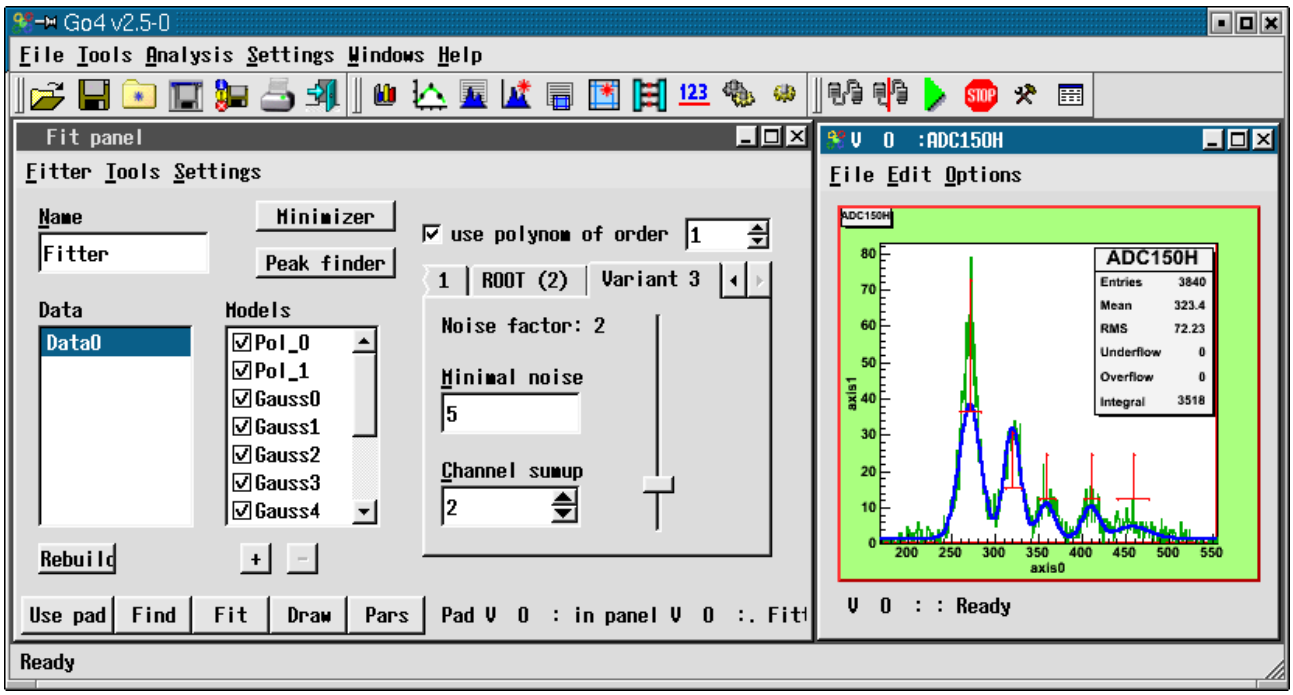
All information of a fit like models (= fit functions) and their parameters, references to the data, and the results are stored in a **fitter object** (=FO). The **fit panel** (activated by button) is the editor of fitter objects. The fit panel is **attached** to a fitter object to edit it. Fitter objects are stored in two different locations:

- Fitter objects can be in the browser (file or memory). By double click the fitter object is displayed in fit panel.
- Fitter objects can be stored in a pad of a view panel (one per pad). Such fitter automatically displayed in open fit panel when pad is activated.

To create fitter for active pad, **Fitter►create for pad** menu item or **Use pad** button of fit panel should be used. The fitter object can always be copied to memory browser and than saved to the file. The data reference of a fit object is changed or set when:

- creating or copying a fitter object to a pad,
- dragging a histogram into a pad (the fitter object of the pad gets the reference to that histogram),
- dragging a histogram name into fit panel.

The next picture shows a pad in a view panel and the fit panel. The peak finder tab is shown.



gui129

On the bottom of fit panel there are five buttons:

Use pad If fitter displayed in fit panel, it will be copied to selected pad in last active view panel. If there is no fitter in fit panel, a new fitter will be created for this pad.

Find Executes peak finder routine. All peak finder parameters should be setup first. Work only in Wizard mode.

Fit Executes fit.

Draw Draw models, backgrounds and model components as sets up in Settings sub-menu.

Pars Show all fitter parameters in a table. Parameters can be listed one by one or in **lines** mode, when one line corresponds to one model and contains amplitude, line position and line width.

There are three different layouts of fit panel, which can be chosen in **Tools** sub-menus:

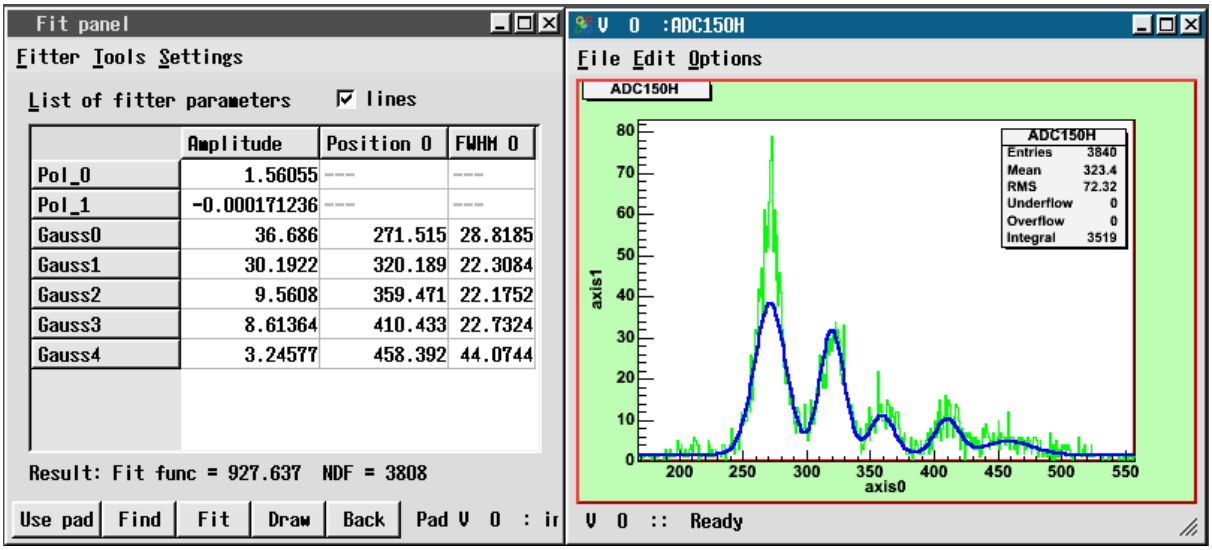
Simple Contains several buttons to fit data to polynomial function, gaussian, lorentz and exponent.

Wizard Intuitive and easy-to-use tool to setup data objects and model components. Also includes peak finder setup. Suitable for most fitting tasks.

Expert Advanced tool, which gives full control over the fitter. Provides a hierarchy view of all objects inside fitter and possibility to change any relevant data fields. Supports all functionality, which may not be presented in Wizard tool.

In wizard mode there are three different peak finders available (see previous figure). Variant 2 is ROOT, Variant 1 searches peaks having specified width range above a threshold, variant 3 searches minima and maxima using a dynamic noise bandwidth. Variant 3 also allows for summing up channels to reduce the noise. Depending on the histogram characteristics, either of these may give good results. One has to play with the parameters. Changing parameters automatically launches a **Find**.

Found peaks are marked in the View panel pad in red. One can move their position and change their width with the mouse. Clicking on a data or model entry the right side of the panel shows related information. Models can be [de]activated clicking on the **OK** boxes or removed by **[-]**. New models can be added by **[+]**. After the fit the results can be seen pressing the **Pars** button (which changes to **Back** to switch the view back):



gui130

Fitter sub-menu has following items:

Create for pad	create appropriate fitter for selected pad in last active preview panel
Delete	delete fitter
Save to browser	save fitter to Go4 memory browser
Update reference	updates references on data objects from file or memory browsers
Print parameters	produces parameters printout, parameters page should be active
Rollback parameters	restore value of parameters, which automatically stored before last fit
Close	close fit panel

Settings sub-menu contains following items:

Confirmation	For each delete action (of fitter, data, model and so on) confirmation message will appear
Show primitives	Show graphical primitives for model position and width and for range settings
Freeze mode	Fit panel is not automatically attached to selected pad, but only by create/copy/move command from Fitter sub-menu
Use current range	At any fit or peak finder action automatically uses range which is currently selected on histogram
Save with objects	Save objects, to which fitter have references, together with fitter. When such a fitter will be loaded, it will have copy of saved objects. Available only in expert mode
Draw model	Draw model of data
Draw background	Draw background (sum of all model components, belongs to background group)
Draw components	Draw all model components, which are not belong to background group
Draw on same pad	Use same pad for drawing or create separate preview panel
Draw info on pad	Draw on pad info box with parameters values
No integral	Do not show any integral values on parameters page
Counts	In lines mode on parameter page additionally shows counts number for every model component inside specified range
Integral	Shows integral value for every model component inside specified range
Gauss integral	Calculates and shows theoretical (based on amplitude and width parameters) integral for one-dimensional gaussian components. None of specified range conditions are taken into account.
Recalc gauss width	For gauss components recalculates sigma values to full width on half maximum (FWHM)
Do not use buffers	Do not use any memory buffers for fit
Only for data	Use buffers only for data objects
For data and models	Use buffers for all data objects and model components
Individual settings	Use buffers as selected individually for each data object and model component

Detailed help on fitter and fit panel can be obtained from the main window **Help►Fit tutorial**.

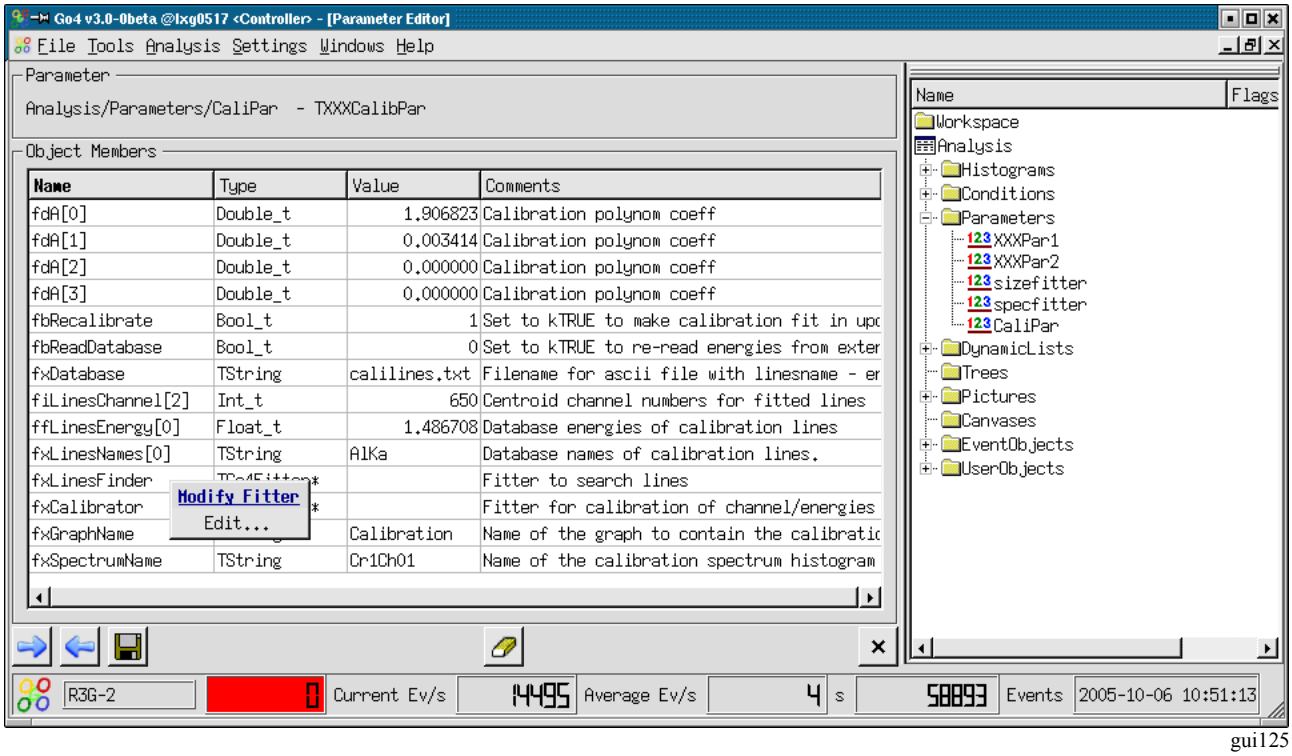
6.11 *Parameters*

6.11.1 Parameter objects

Parameters are objects containing a user defined structure of values. These can be applied for controlling and calibrating the user analysis apart from the analysis framework configuration. All user parameters should be subclasses of *TGo4Parameter*. They can be created in the user analysis code and are registered to the Go4 framework by method *AddParameter(TGo4Parameter* mypar)*. Once a parameter was registered, it appears in the Go4 **Parameters** folder, it is saved and can be restored from the auto-save file, and it can be edited and updated from the Go4GUI by means of the parameter editor.
Note that the Go4 GUI has to load the libGo4UserAnalysis.so to edit and save any user defined parameter object from the analysis.. See 6.2 how to load libraries to GUI.

6.11.2 Parameter editor

Double clicking a parameter icon 123 in the browser will open the parameter editor as seen in the picture. All known members of the user parameter class and its base classes are shown here with their names, their type and their current value.



gui125

Currently supported types are:
all basic signed and unsigned types, e.g. `Double_t fdEnergy; Bool_t fbIsOK;`
the ROOT *TString* class to wrap text strings, e.g. `TString fxMyFilename;`
pointers to *TGo4Fitter* objects, e.g. `TGo4Fitter* fxUnpackfitter;`
and arrays of these in 1 or 2 dimensions, e.g. `UInt_t fuVal[42]; Float_t ffVoltage[5][100];`
Comments behind member declarations are shown in the **Comments** column.
Aggregations and **pointers** to basic types are **not supported** at the moment (except for aggregated fitter objects).
Arrays of data are expanded and collapsed in the table by double clicking on the array name. Additionally, the right mouse button will open a popup menu to navigate through the array without expanding it completely.
The values of the data can be edited after double clicking in the value field of the data member table. Note that any editing action has to be finished by pressing “return”, “tab”, or “cursor” before it is valid. To apply the changes, press which will update the edited parameter on the analysis side. This is done by method *UpdateFrom(pointer to new)* provided by the user class. This means that arbitrary functions can be executed! The changing of data members is fully controlled by the user class. Vice versa, will refresh the table shown in the editor from the current values of the analysis parameter. Note that all changes not yet applied to the analysis or saved are overwritten on refresh!
If one is working on a parameter loaded from a file, button will appear instead of , doing a refresh from the source file. Note that the original parameter in the file is not changed by the editor immediately; the root file is updated only

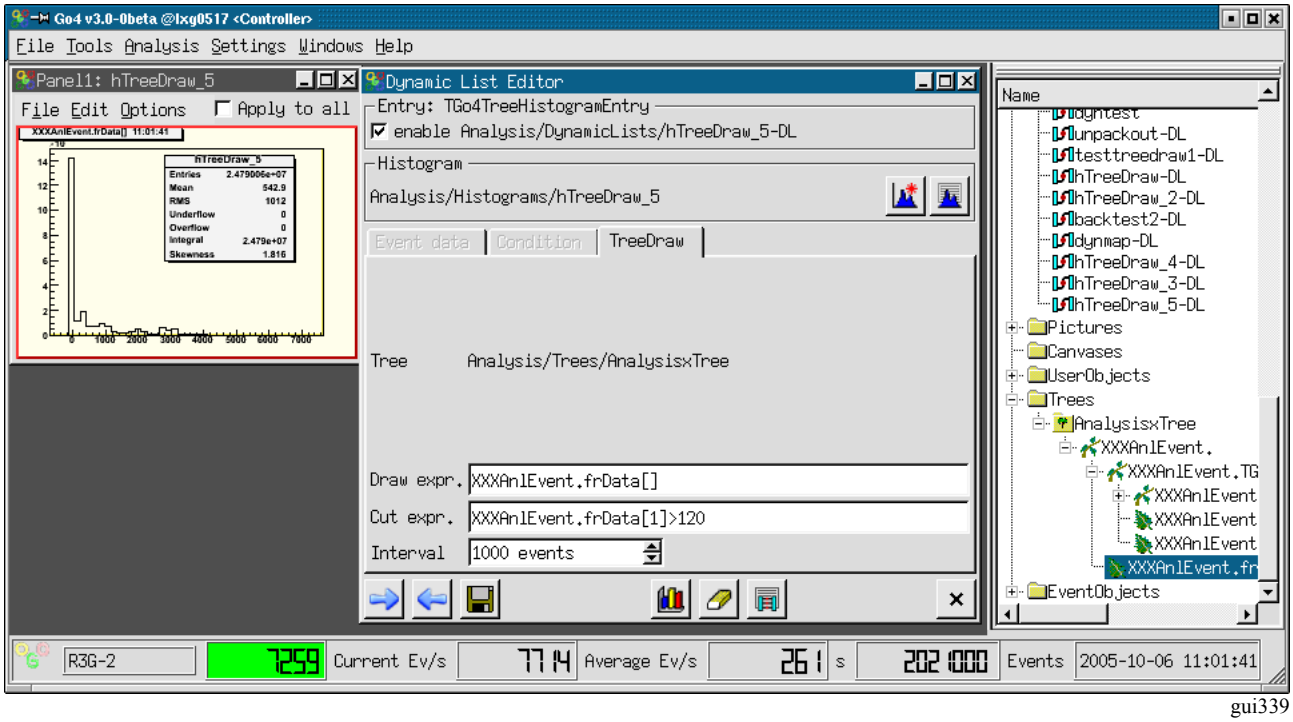
when using the save button . Then a save dialog window will appear, that allows either overwriting the original parameter, or saving the changed object to another file.
Finally, will erase all editable fields of the table. will close the editor without modifying the analysis parameter.

6.11.3 Parameters containing fitters

Sometimes it might be useful to exchange a Go4 fitter object between the analysis and the GUI. A fitter, e.g., may be prepared using the FitGUI and then sent to the analysis client where it can be applied to some histograms during analysis. Vice versa, one might want to display the resulting parameters of automatic fits in the analysis on the GUI. Therefore, the Go4 parameter concept supports the *TGo4Fitter* class as aggregation member, i.e. a pointer to a fitter can be accessed by means of the parameter editor.
The Go4 framework already offers the parameter class ***TGo4FitterEnvelope*** that contains one fitter object. This fitter may be accessed in the analysis by method *GetFitter()*. In this case it is important that the fitter object itself is exchanged inside the parameter each time the parameter is updated. Thus the user should not keep the pointer to the fitter in his/her analysis class, but request the fitter from the (persistent) *TGo4FitterEnvelope* parameter with the getter method when the fitter should be used.
Additionally, **any user defined subclass of *TGo4Parameter*** may contain references to several fitters or even arrays of fitter references. Here it is the user responsibility how the fitters refresh their settings in the *UpdateFrom()* method. Moreover, one may implement getter and setter methods for the most important values of the fitters without the need to access the internal fitters directly. An example is *TXXXCalibPar* in the `Go4Example2Step` directory.
Pressing the right mouse button over the name of a fitter member will open a **context menu**. Selecting **Edit...** will open the Go4 FitGUI window (see chapter 6.10, page 70). A copy of that fitter is put into the local workspace of the Fit GUI to be edited or to be applied on any histogram. Selecting **Update fitter from fitgui workspace** in the context menu, the fitter in the parameter object is replaced by a copy of the fitter that is currently active for the Fit GUI. So any fitter existing on the GUI may take the place of any fitter inside a parameter. Note that the original fitter member in the parameter will be lost after this action unless it is refreshed by from analysis again! To send the changes in the fitter back to the analysis client, like for all parameters the button must be pressed.
Note that in case of a fitter pointer array (e.g. `TGo4Fitter* fxFitters[10]`), the context menu will show both the items to manipulate the array view and to edit or update the selected fitter.

6.12 *Dynamic lists*

The Go4 dynamic list is a mechanism to connect the event data with a histogram and a condition. The histogram is filled from certain data members of the event during the analysis. Optionally, the histogram may be filled only if a condition that is tested against other data members of the event is true. In contrast to the histograms filled from the compiled user analysis code, the dynamic list offers the possibility to define these relations on-line during the running analysis. The dynamic list and all newly created histograms and conditions may be stored in the Go4 auto-save file and are recovered on the next analysis initialization (or **Submit** button in the configuration menu).
In the Go4 browser, the dynamic list folder contains all existing dynamic lists (currently only one default list). Each list shows the existing dynamic entries by name. Double clicking on a dynamic entry will open the dynamic list editor to display and change it.
To create a new dynamic entry, button of the main window tools menu will open the **create new entry** dialog window. Here you can define the name and the kind of the dynamic entry. There are 2 different kinds of Go4 dynamic entries: The **TreeEntry** and the **PointerEntry** (see below). After pressing "Create remote" button, the new dynamic entry will appear in the browser in analysis subfolder DynamicLists.
To delete a dynamic entry completely, select its icon in the Go4 browser and select in the right mouse button menu.



gui339

6.12.1 Dynamic list editor

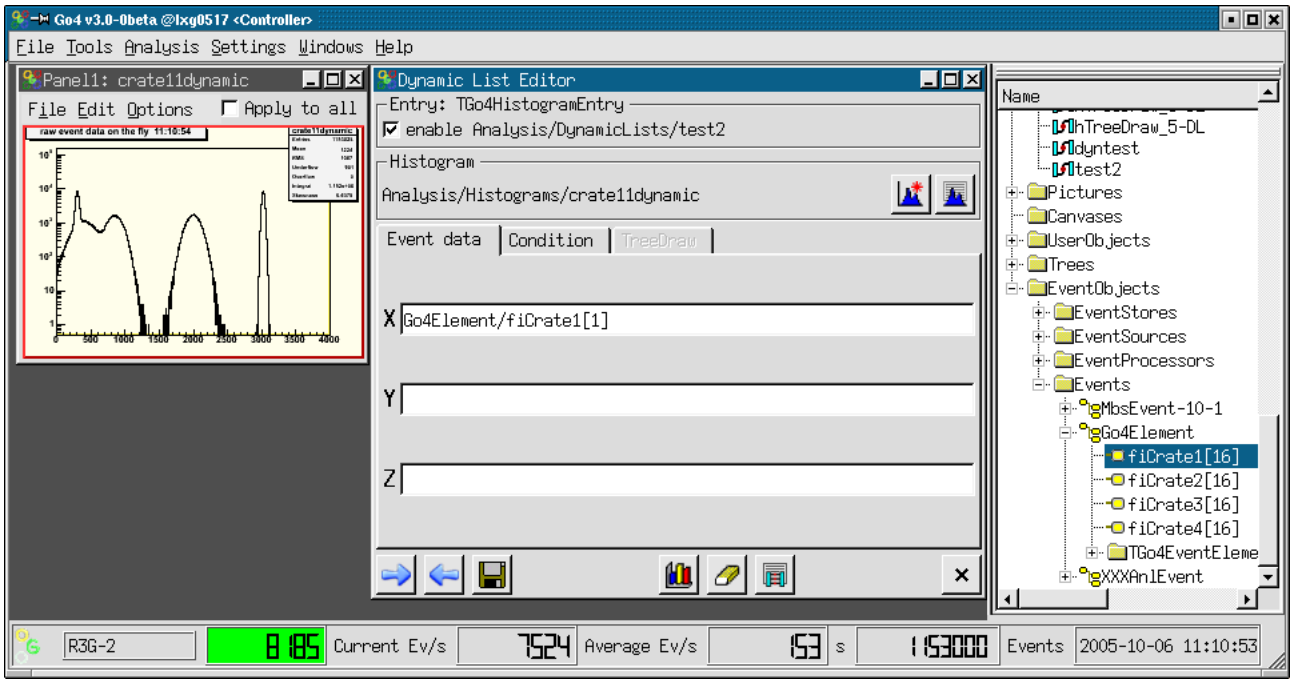
Depending on the kind of the entry, different sub-pads of the editor are enabled: The **Histogram** and **TreeDraw** sub-pad for the **TreeEntry**, and the **Event data** and **Condition** sub-pad for the **PointerEntry**, respectively.
Any dynamic entry can be enabled or disabled by switching the **enabled** checkbox. A disabled entry will not be processed, but is still in the dynamic list. Note that if a dynamic entry fails on initialization (e.g. unknown object names), it is disabled automatically.
To apply the changes, press which will update/create the edited entry on the analysis side, respectively. Vice versa, will refresh the values shown in the editor from the current status of the analysis dynamic entry. Note that all changes not yet applied to the analysis are overwritten on refresh! A label will appear near the update button if the changes have not been applied to the analysis yet.
If one is working on a dynamic entry loaded from a file, button will appear instead of , doing a refresh from the source file. Note that the original dynamic entry in the file is not changed by the editor immediately; the root file is updated only when using the save button . Then a save dialog window will appear, that allows either overwriting the original parameter, or saving the changed object to another file.
will clear the target histogram in the analysis to zero counts, and will reset the events in the backstore tree (in case of tree draw entry, see below.) This allows to observe changes of the dynamic entry setups directly if the target histogram is monitored. will close the editor without modifying the entry.
The editor offers the additional feature to get some information of the histogram and condition status from the analysis. Clicking in the Histogram or in the Condition sub-frames will retrieve and display the current object status in the

histogram or condition status windows, respectively (see chapter 6.13, page 77). This may be useful to check if histogram or condition settings (dimension, ranges, bin size, etc.) are suitable, without requesting these objects in the browser. Additionally, some filling and testing statistics is shown here. The GUI tool tips show brief explanations for each information line.
The button prints the names and connections of all existing dynamic entries to the analysis output window.
New histograms or conditions may be created in the analysis by the or the button, respectively. For histograms, the standard histogram creation window (see chapter 6.6.3, page 56) pops up. Use the **Create Remote** button here. For conditions, the "new condition" dialog is started (see chapter 6.8.5, page 67).

6.12.2 Entry for tree draw

Go4 uses the ROOT *TTree::Draw()* mechanism for the on-line evaluation of the data. This works just as described in the ROOT users Guide: A string expression defines which leafs of the tree shall be scanned by name. Additionally, the name of the output histogram must be specified; the histogram may either already exist (**Create Remote** from Go4), or it is created from the first *TTree::Draw()* by ROOT with automatic range and binning. Instead of a Go4 condition, this mode works with a *TCut* string expression to filter the histogram filling.
Note that the *TTree* must exist for this mechanism. Usually, the *TGo4FileStore* output will create and register a tree that can be used here. If no file output is needed, one can switch on the **TGo4BackStore** output (configuration window) which will fill a temporary *TTree* in memory that is cleared after each *TTree::Draw()* scan of the dynamic list. The *TTree::Draw()* is not performed for each single event, but after a number of events have been filled into the tree. This number can be specified in the user analysis by *TGo4Analysis::SetDynListInterval(Int_t val)* or by the **Interval** field.
A new tree draw entry can be created either from the Go4 tree viewer (drag of the tree name from the Analysis browser and press), or from the **Create Dynamic Entry** dialog. In the latter case, the tree name, the histogram name, the draw expression and optionally a cut expression may be specified directly in the dynamic list editor after creation. This works by "drag and drop" of historams and tree leafs from the browser to the corresponding fields of the dynamic list editor. Note that the *TTree* name is recognized automatically from the dropped leaf.
The advantage of a tree draw entry is that it can access any level of substructures of the event if it is resolved in the *TTree* (depending on split level); the Go4 composite event data may be fully accessible here. It offers all functionality of the ROOT *TTree::Draw()*. The disadvantage is that you need to fill the event data into a tree to access it. The histograms are not filled event by event, but the tree is processed in event buffers. The buffer size should be adjusted by the user depending on the typical event rate. Since the pointers to the data and the histogram are searched by name for each *Draw()* call, the performance is slow compared to histogram filling from direct pointer access like in the precompiled user analysis case.

6.12.3 Entry for event loop



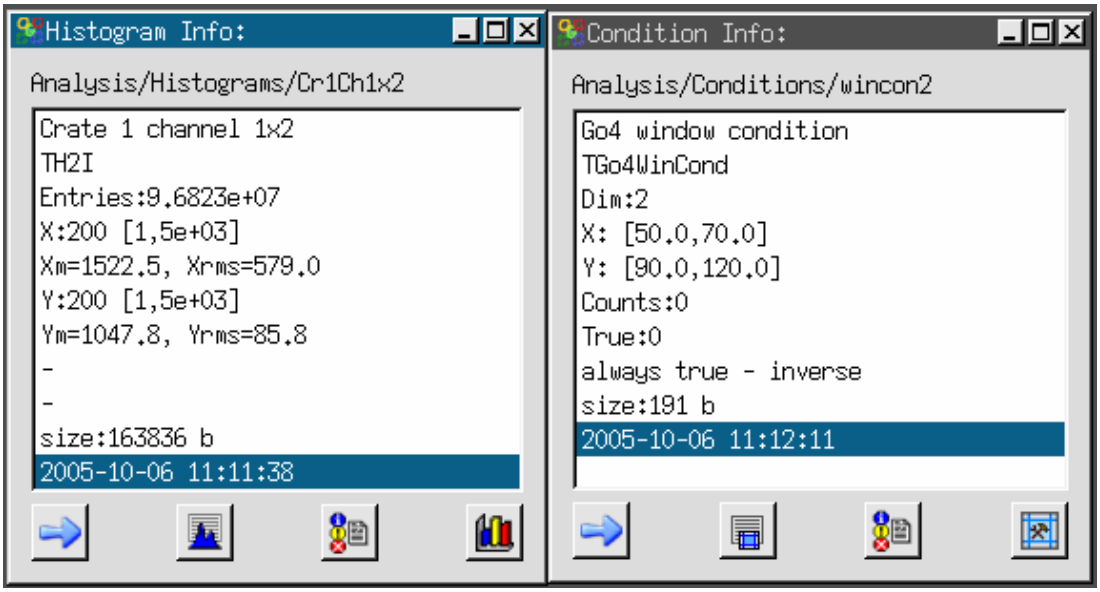
gui340

In this mode (**PointerEntry**), the pointers to histogram, event data and an optional Go4 condition are looked up by name once on initialization of the dynamic list. During the analysis, these pointers are used directly to test the condition and

fill the histogram event-by-event. The information to locate the pointers is taken from the ROOT *TClass* information of the user event classes; it is not necessary to fill the event into a *TTree*.
For the pointer entry, at least the name of an existing histogram and one dimension of the event data must be specified. This is done in the **Event data** tab of the editor. Usually, for a new pointer entry the histogram should be created by  (see above). The new histogram item must then be dropped from the browser to the dynamic list window.
The event data is defined by the event name and the name of the data member of the corresponding event class, separated by a slash ("/"). The Go4 browser Analysis folder offers a view of all existing *TGo4EventElements* in the **EventObjects.Events** folder. From here you may just drag and drop the **Data member** item to the corresponding field of the dynamic list editor. Note that data arrays are shown with their maximum size here, you need to edit the index afterwards to specify the desired array member.
Similarly, the data to test the condition can be defined in the **Condition** tab of the editor. The condition is usually created and registered in the compiled user analysis and is identified by name here. Polygon conditions and 2 dimensional window conditions need the event data specifications both in x and y directions. Note that the condition event data lines should be left blank if the condition shall not be tested in the dynamic entry (i.e. the histogram is filled anyway). With  a new condition can be created. Button  will open the editor for the specified condition.
The advantage of the pointer entry is that you do not need a *TTree*. Testing and filling is done for each event by pointer without any additional string compare after initialization. Therefore it is faster than the tree draw. The disadvantage is that currently only one level of substructures and only one dimensional arrays are supported (to be improved...). Implicit summing up of not specified array indices, like in the *TTree::Draw(),* is not possible here.

6.13 *Histogram/condition information*

To check the properties of a histogram or condition, general property windows exist for these objects. They support drag and drop of icons from Go4 browser. These windows will also pop up from the browser's context menu when the button is chosen.

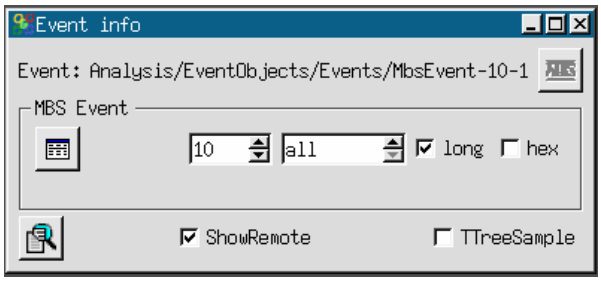


gui341

With the button or the button of the tools menu one opens the histogram or condition information window, respectively. To see the properties of a histogram or condition, drag the icon from the browser into the window. With the information is updated from analysis. With the information is output to the GUI start up window, or into a log file if specified in the log settings (see chapter 6.1.2, page 43). With all histograms (all conditions) are listed in the analysis output window. starts the condition editor for a condition. displays the histogram in a view panel.

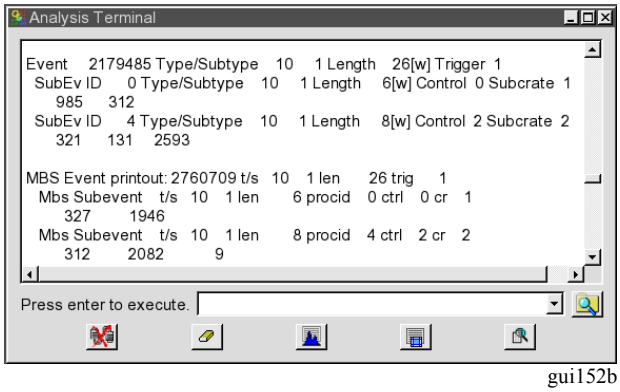
6.14 *Event information*

The event information tool window allows to control printout of event samples from the analysis. The button of the tools menu will open the event information window. This button is also available as a shortcut in the Qt analysis terminal. The entry of the browser's context menu (right mouse button) over an event itemwill open the event information tool, too.



gui342

The name of the examined event is shown in the top text line. By default, the MBS event is chosen for printout. The event object names may be dragged and dropped to the event information window from the Go4 browser. Clicking the button will switch to the MBS event mode directly without the need to drag the **MbsEvent-10-1** icon.



gui152b

The **ShowRemote** checkbox selects if the printout of the event sample is done in the remote analysis terminal, or in the terminal where the GUI was started. The **TTreeSample** checkbox selects if the *PrintEvent()* method of the event shall be called (**TTreeSample** off), or if the sample event shall be written to a ROOT Tree which will use the *TTree::Show()* method to scan and display the data (**TTreeSample** on). Note that for user event classes that do not implement a *PrintEvent()* nothing will be displayed except for the **TTreeSample** mode.

Each click on button will print events as shown in the upper part of the screen shot left side. The examine button will display a new printout of the currently

active event (lower output on the left). Note the different format!
Additionally, for MBS events this window provides in the **MBS Event** sub-panel parameters for the *SetPrintEvent()* method. One can specify in the left field how many MBS events arriving shall be printed out in a special format. In the next field a sub-event id may be filtered (default is to display all sub-events). The **hex** checkbox selects to print the sub-event data either in hex or in decimal format, while the **long** checkbox defines if the data is seen as longwords or words. Pressing the button will resubmit these settings to the analysis thus initiating a new printout of n events. Note that the **MBS Event** sub-panel is independent of the settings for the regular printout of the current event. It works for the remote analysis terminal only, and it uses a different printout format than the *TGo4MbsEvent::PrintEvent()* or *TTree::Show()* methods.

6.15 *Hot start*

When starting the GUI several actions have to be done to get the analysis running. If these actions are always the same it would be convenient to save them in a file and execute this file when starting the GUI next time. This mechanism is called hot start. The typical actions are:

- Launch analysis client
- Submit analysis configuration
- Get analysis folders by
- Set histograms and pictures into monitoring state
- Open some view panels and display histograms or pictures

After GUI and analysis are configured, one can create a hot start file by **Settings►Generate hotstart**. A file selection menu pops up were one can specify a file name. The postfix should be `.hotstart`. The next time one can start the GUI with this filename as argument (`.hotstart` can be omitted). Then all actions stored in the file are executed.
With care, this file could even be edited.

6.16 *User GUI*

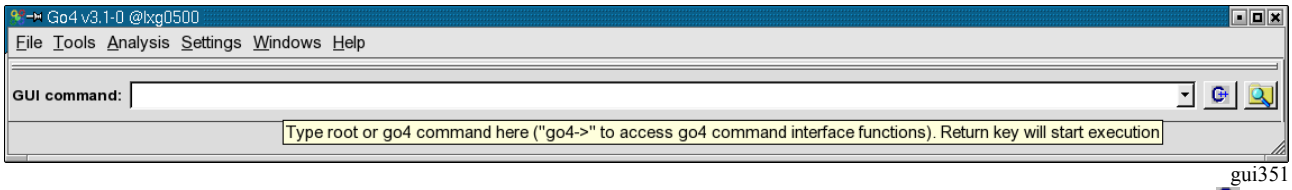
Go4 provides a possibility to execute user widgets on GUI side. There is an example of a user GUI, included in the standard Go4 distribution in directory `$GO4SYS/Go4UserGUI`. It can be activated by pressing button in **Tools** of main window.
The easiest way to create a user GUI is to copy the content of the standard example to another directory (e.g. ~/UserGUI) and compile it there (`make clean, make all`). The user should also specify the path to this directory in `GO4USERGUI`

```
export GO4USERGUI=~/UserGUI
```

The `GO4USERGUI` variable can also include the name of the library (default `libGo4ROOTUserGui.so`) which is loaded when user GUI is started. This library must include the special function *StartUserGui()* which loads the qt widget library (default `libGo4UserGui.so`) and creates the top level widget of user GUI. At the next start of the Go4 GUI pressing the specified GUI will be opened.
The user can freely modify any widgets in the example and create new ones. Changes in library names or the top widget class should be reflected in the `GO4USERGUI` variable and the *StartUserGui()* function.
There is a support of "old style" user GUI, created with older version of Go4 (up to v2.8). In that case correct path to libraries should be specified like:

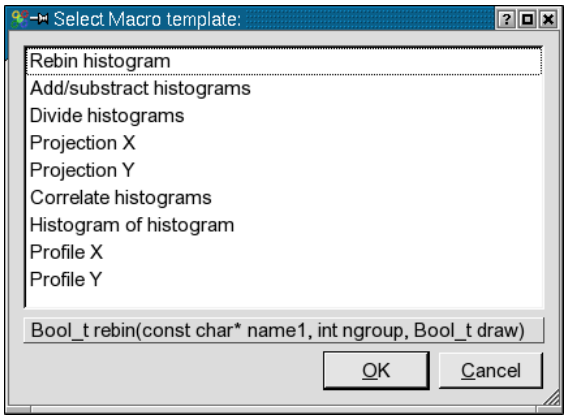
```
export LD_LIBRARY_PATH=~/OldUserGUI/Go4Library:$LD_LIBRARY_PATH
```

6.17 *Macro execution in GUI*



gui351

The command line window can be enabled with the settings pull down menu like all other windows. With button the following menu appears with a list of provided macros. These can be executed directly or can be used as templates for other macros. An environment variable **__GO4MACRO__** is defined and can be used to write macros to run in several environments: plain ROOT , Go4 GUI, or analysis (see 6.4.3, page 49). The provided macros `hishisto.C` and `corrhistos.C` are written this way. The histograms to be processed are accessed through the Go4 browser or from a file, respectively. With the file browser button the current directory is searched for macros.



gui350

`corrhistos.C` takes the bin contents of two histograms and creates a two-dimensional graph.
`hishisto.C` creates a histogram and makes a distribution of the bin contents of the source histogram.
An interface is provided to access objects from the Go4 browser. A description can be found in the reference manual.

Caution! Macros running inside the GUI can crash the GUI!

`$GO4SYS/macros` directory should be added to entry `Unix.*.Root.MacroPath` in `.rootrc` setup file.

7 Analysis Server for ROOT macros

The Go4 analysis server offers the possibility to observe and control execution of normal ROOT macros from the Go4 GUI. This allows the development of analysis code without respect of Go4 analysis framework classes (like *TGo4EventProcessor*, *TGo4AnalysisStep* and so on) still providing remote access to the running environment of a user analysis.
It is possible with minimal effort to observe histograms, produced and filled by practically any running ROOT script. The script `go4Init.C` initializes Go4 and starts the analysis server in background. Function `go4RegisterAll()` then scans the current directory for existing histograms and makes them available remotely.
Usage:

1. To enable ROOT to find the go4 macros one should enter in the .rootrc a line
   `Unix.*.Root.MacroPath: .:$(ROOTSYS)/macros:$(GO4SYS)/Go4AnalysisClient`
   (Note that `.rootrc` may be in current directory or in $HOME.
   The standard provided by ROOT is in `$ROOTSYS/etc/system.rootrc`)
2. Run normal ROOT session.
   Execute `go4Init.C` script by command:
   `root [0] .x go4Init.C`
3. Run user script:
   `root [1] .x userScript.C`
4. When `go4Init()` is executed, go4 will start the server and printout the port number for connection:
   `"Waiting for client connection on PORT: 5000"`
5. Start the Go4 GUI in and connect to the analysis server running in the CINT. See section 6.3.2 page 46 for more.

The Go4 framework can be accessed after `go4Init` by the global method
`TGo4Analysis* go4= TGo4Analysis::Instance();`
After this call, variable `go4` can access any method of the analysis framework.

7.1 *Methods for object registration*

Any object to be seen remotely by the GUI must be registered by one of the following methods:

- `go4->AddHistogram(his);` // makes histogram `TH1* his` available in the Go4 GUI
- `go4->AddAnalysisCondition(conny);` // dito for TGo4Conditions
- `go4->AddParameter(par);` // dito for TGo4Parameters
- `go4->AddPicture(pic);` // dito for TGo4Pictures
- `go4->AddTree(mytree);` // register TTree, but do not change Tree ownership to Go4
- `go4->RemoveTree(mytree);` // unregister TTree: important to cleanup reference in Go4 if tree // is removed from ROOT (closing TFile !)
- Please see Go4 Reference Manual for other available `Add...` methods!

The `go4RegisterAll()` function (from Go4Init.C) registers all histograms found in the current directory. Some more information can be found in the example macros (see below).

7.2 *Methods for run control and execution*

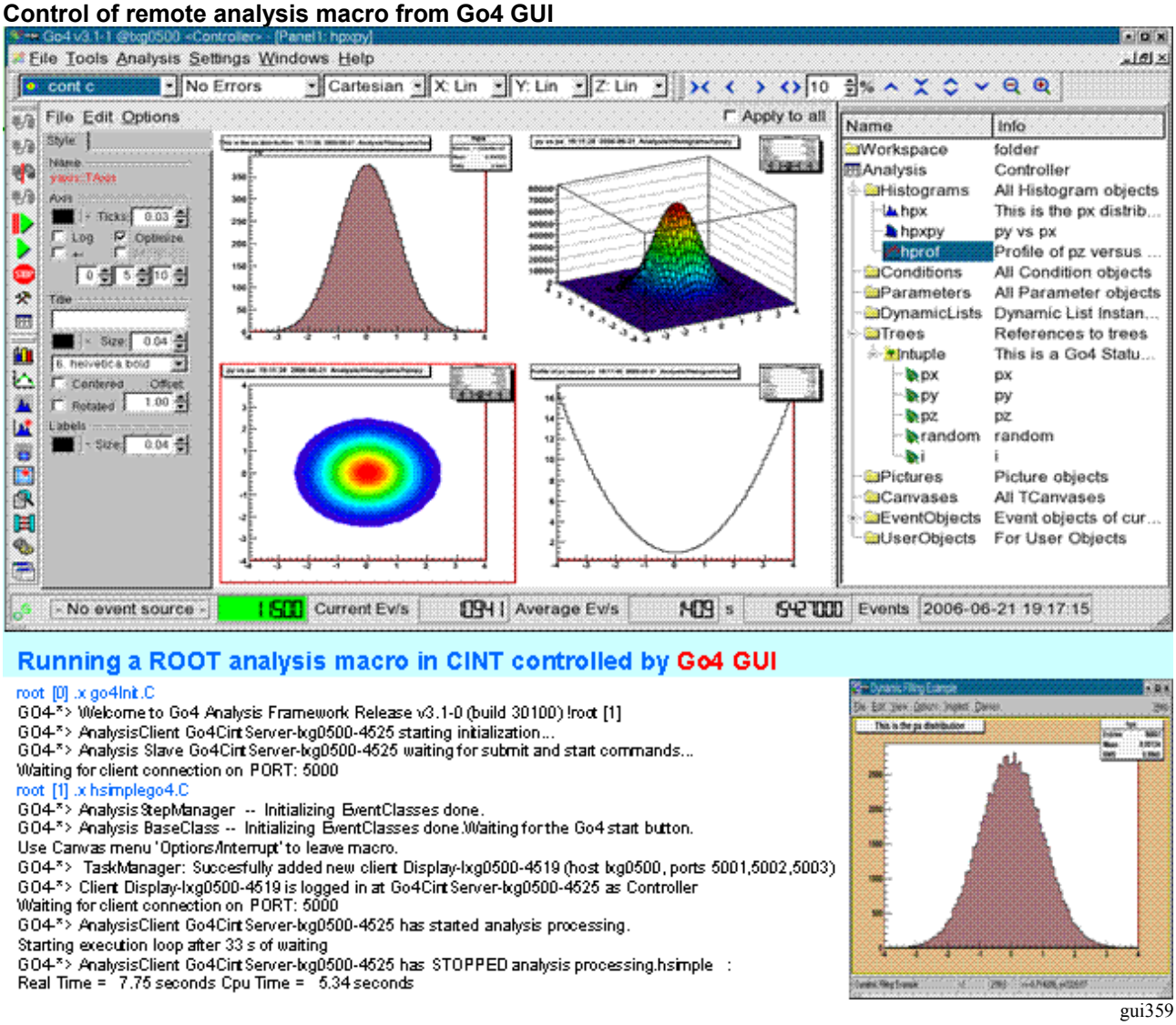
- **`Int_t seconds=go4->WaitForStart();`** Polls until the Go4 is set into the "running" state (by **Start** button on GUI or `SetRunning()` method) with 1 second interval. Returns number of seconds from begin of wait until "running" is switched true. If negative value is returned, a ROOT interrupt has happened during wait (e.g. Ctrl-C on CINT Canvas).
- **`Int_t state=go4->Process();`** Process one main cycle of Go4 event loop from macro. Will first execute any command from GUI, second call the Go4 main cycle to process analysis steps, user event function and dynamic list (if existing). **This call is required inside any explicit loop in the macro to process go4 framework analysis actions.** The GUI event rate meter is also updated by this method. Return value is <0 if running state is stopped, otherwise 0.
- **`go4->SetRunning(Bool_t on);`** Switch Go4 running state from inside a macro. Useful to react on analysis conditions
- **`Bool_t on=go4->IsRunning();`** Check the running state of the Go4. Maybe obsolete since this is done implicitly in methods *WaitForStart()* and *Process()*. However, macro loop may be controlled from GUI independent of Go4 main loop processing.

7.3 *Examples:*

The following examples can be found in $GO4SYS/Go4AnalysisClient package. It is recommended to copy these macros to a user directory with write access, before executing them.

- **hsimple.C** This is a standard ROOT example from $ROOTSYS/tutorials. The only modification is to call `go4RegisterAll()` after creating histograms.
  To run this example, start a regular ROOT session, init the Go4 server and execute script:
  ```
  root [0] .x go4Init.C
  root [1] .x hsimple.C
  ```
- **hsimplego4.C** A variation of `hsimple` example. This macro will wait until the Go4 start button is pressed and then run the random filling in infinite loop (mind your disk space, since a *TNtuple* is filled into a file here!) Registered objects may be monitored. The loop can be started and stopped at any time from the Go4 GUI. Please try the remote tree draw on the *TNtuple* from the Go4 GUI and view the newly created histograms. Try to launch the *TBrowser* before executing the macro and inspect the content of the "Go4" folders locally...
- **treedrawgo4.C** Macro works on tree in a file. As before, first execute .x go4Init.C:
  ```
  root [0] .x go4Init.C
  root [1] .x treedrawgo4.C("filename")
  ```
  The "*filename*" specifies a ROOT file "*filename.root*" that contains a *TTree*. Note: first tree found in file will be used.
  This macro contains 2 examples on trees:
  1. Direct *TTree::Draw()* expressions are executed; after finishing, a message is sent to the Go4 GUI and the output histograms may be viewed here.
  2. After registration of the *TTree*, the *go4->Process()* will be executed in a loop. Please try the remote tree draw on the *TTree* from the GUI and view the result histograms. Loop may be controlled by the **Start/Stop** buttons as in example `hsimplego4.C`.

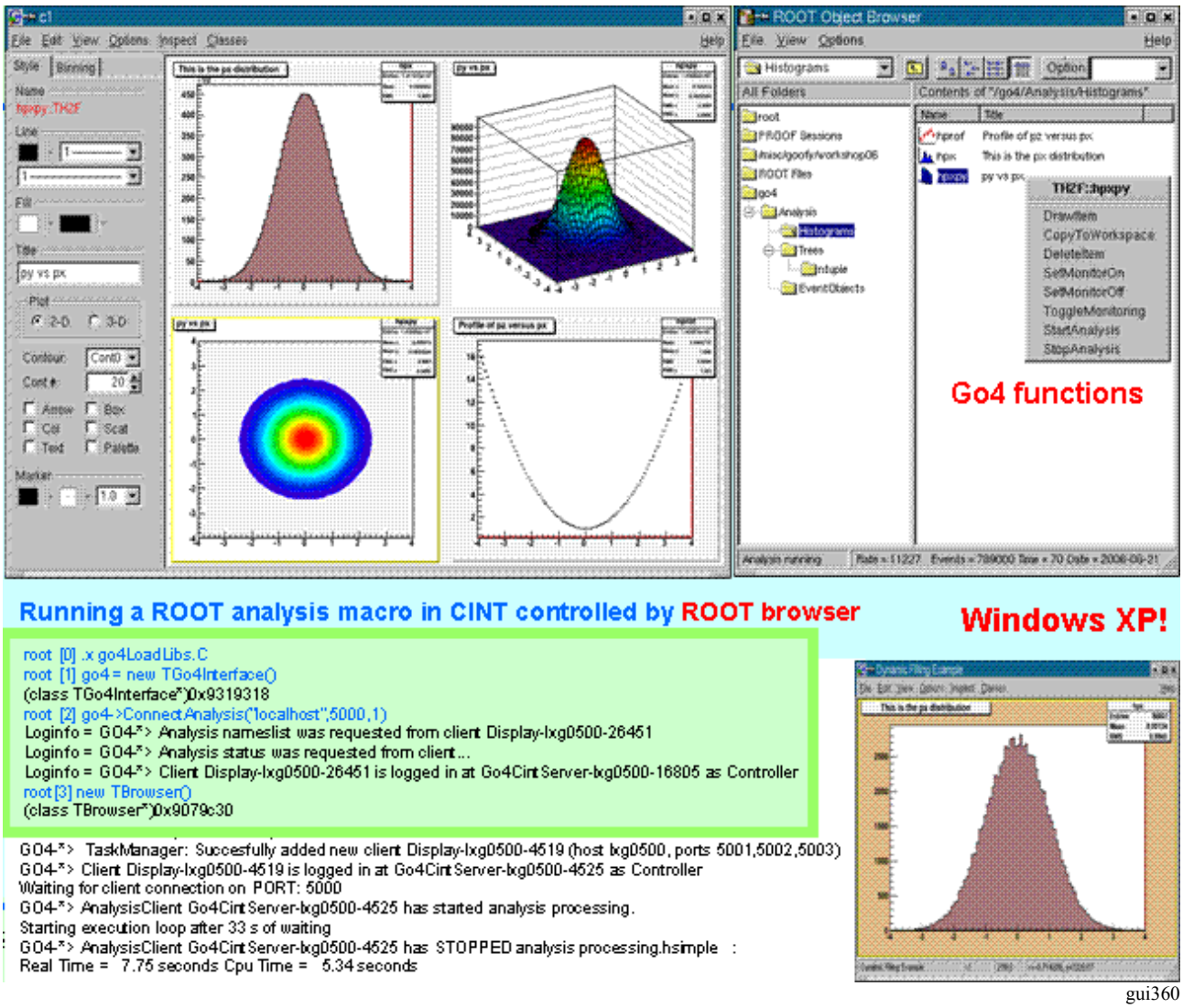
Control of remote analysis macro from Go4 GUI



gui359

8 Control of remote Go4 analysis from a ROOT session

Besides the full featured Qt GUI, the Go4 analysis may be controlled and observed by a regular ROOT CINT session, using the native ROOT GUI for display.
The following screenshot shows at the bottom a go4 CINT analysis server task. Here example hsimplego4.C is running (see 7.3). This process is connected with the ROOT session in the upper part of the picture, which uses the regular ROOT GUI to browse and display the analysis objects. This is just like it would be possible with the usual Go4 GUI. Actually, a multithreaded Go4 master task is running in the background of the upper ROOT session, while a Go4 slave task is working on the analysis in the lower root session. This analysis process may not only be a root session with Go4 analysis server, but may as well be a compiled Go4 analysis client executable (MainUserAnalysis).



gui360

8.1 *Initialization*

The controlling Go4 master process is realized in the ROOT session by the **TGo4Interface** class. After starting a regular ROOT and loading the Go4 libraries, the call

```
root [0] new TGo4Interface
```

will instantiate the master task framework. Explicit loading of libraries is not necessary if the corresponding ROOT mapfile mechanism is used. Once initialized, the variable `go4` is defined as a pointer to the interface instance and may use all methods of class `TGo4Interface`. **Note that in the analysis server session as described in section 7, variable `go4` refers to the class `TGo4Analysis` instead!**

8.2 *Connecting the analysis*

To connect to an existing **analysis server**,use

```
root [1] go4->ConnectAnalysis("localhost",5000,0, "XXXview");
```

Arguments are: hostname of the server, the port number, the login account (0=observer, 1=controller, 2=administrator), and the password. If password is left out, the default password of this account is used.
Alternatively, an analysis client may be started from this session using

```
root [1] go4->LaunchAnalysis("test","/u/user1/go4",
                             "MainUserAnalysis","lxi003");
```

With arguments: arbitrary name ("test"), path to the analysis executable, name of the analysis executable, and node where analysis process shall be started.
The above methods correspond to the **Connect analysis** and **Start analysis** dialogues of the Go4 GUI (section 6.3).

8.3 *Controlling the analysis by command*

Once the connection to the analysis process is established, it can be controlled by several methods:

- **`go4-> SubmitAnalysisConfig();`** Submit the analysis configuration. This corresponds to the **Submit** button of the Go4 GUI. Usually, the configuration is retrieved from analysis after connection. It may be modified by several methods of the TGo4Interface before submit, or it may be submitted unchanged. A submit is required in any case before analysis can be started. Note that this command is not allowed when logged in as observer.
- **`go4-> StartAnalysis();`** Start the analysis run. This corresponds to the **Start** button of the Go4 GUI. Note that this is not allowed when logged in as observer.
- **`go4-> StopAnalysis();`** Stop the analysis run. This corresponds to the **Stop** button of the Go4 GUI. Note that this is not allowed when logged in as observer.
- **`go4-> StartMonitoring(Int_t period=10);`** Start monitoring all objects that are set to monitoring state and drawn. The update period can be specified in seconds.
- **`go4-> StopMonitoring();`** Stop monitoring all objects. Will not reset the monitoring property of the objects.
- **`go4->DisconnectAnalysis();`** Remove connection to remote analysis process.

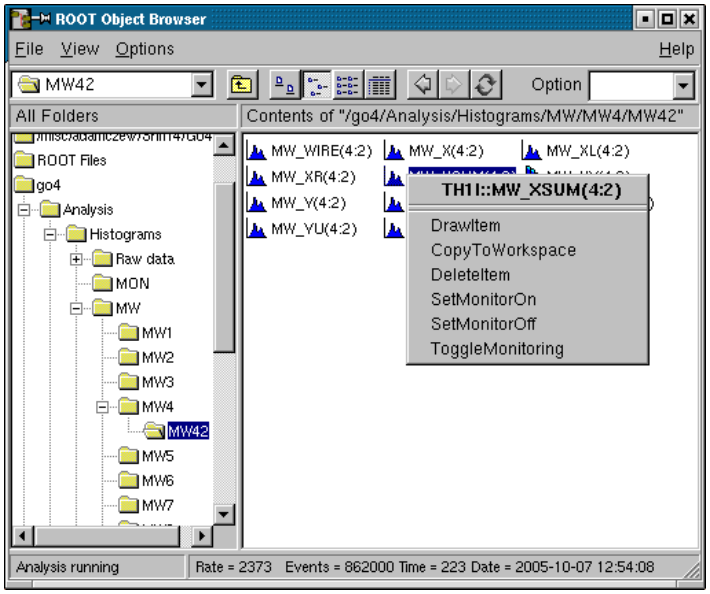
For a complete reference of available methods, please see the header file $GO4SYS/include/`TGo4Interface.h`.

8.4 *TBrowser extensions*

In addition to the analysis control by TGo4Interface calls, the regular ROOT browser will offer some extensions after the connection has been established. Start the browser with:

```
root [2] TBrowser br;.
```

If connected to the analysis, there is a Go4 folder among the regular ROOT folders. This will browse the structure of the remote analysis with subfolders and all objects. Both histograms and Go4 pictures may be drawn to a new canvas by double clicking on the item. Go4 conditions will be drawn on double-click only together with the histogram that was bound to it. The ROOT right mouse button menu has entries added for the remote Go4 objects:



gui343

- **Draw Item** - will draw it if possible, just like double-click
- **Copy to Workspace** - Produce fix copy to the Workspace folder in local memory. Just like in the regular Go4 GUI.
- **Delete Item** - remove object from analysis if possible
- **Set Monitor On/Set Monitor Off** - Switch the monitoring property of the selected object
- **Toggle monitoring** – Start and stop monitoring in general. A dialog will appear to request the monitoring periods in seconds. For zero period, monitoring will be stopped. This corresponds to TGo4Interface methods StartMonitoring() and StopMonitoring().

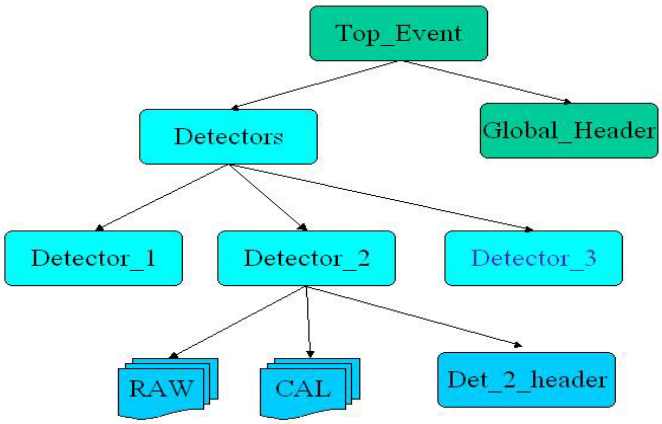
The status line at the TBrowser bottom will show the analysis rate meter, and eventually some messages retrieved from the analysis. Additionally, the status messages are printed out to the CINT terminal.

9 The Go4 Composite Event Classes

9.1 *Introduction*

A "real" example of the composite event usage can be found on the Go4 web.
Thinking about possible structure for typical experimental data, one naturally comes to something more or less close to the following scheme:

- A so-called event model should be defined as the unit for the data processing. It represents the record of all physical interactions in the detector after the reaction between a beam particle and a target. This event can be real, calibrated or simulated as one process real, calibrated or simulated data. The event can contain the original data from the different detectors (Raw level) but also reconstructed data resulting from the different reconstruction steps (Calibration, Hits, Tracks ...) As it contains the complete set of physical data, the event is a complex object. To avoid having a monolithic event containing everything on the same level, one can take advantage of structuring it into smaller independent components (sub events). hangs are easier to perform, extensions are easier to integrate and your design is much easier to understand. Separating the whole into parts is also an advantage if one considers using partial IO functionality: one should be able to retrieve only some part of the whole event from the file i.e. the relevant part for his analysis, the rest being deactivated in the IO.
- The decomposition of the event should follow fixed rules. For example, one can decide to have the scheme as shown in the picture



- In this example the top-level event is composed by a global header containing for example the general setup of a typical experiment i.e. RunID, beam energy, magnetic field value, target type and size etc .... On the same depth-level a detector folder contains the different detector's specific data i.e. Raw , Cal, Hit,... data-levels and eventually a specific detector header.
- It should be possible to access directly a part in the tree-like structure of an event. For that purpose each component should be uniquely identified.
- The event is obviously a persistent object. Therefore the data should be organized into a ROOT tree with a branching layout which fit to the natural branching of the tree-like event structure.

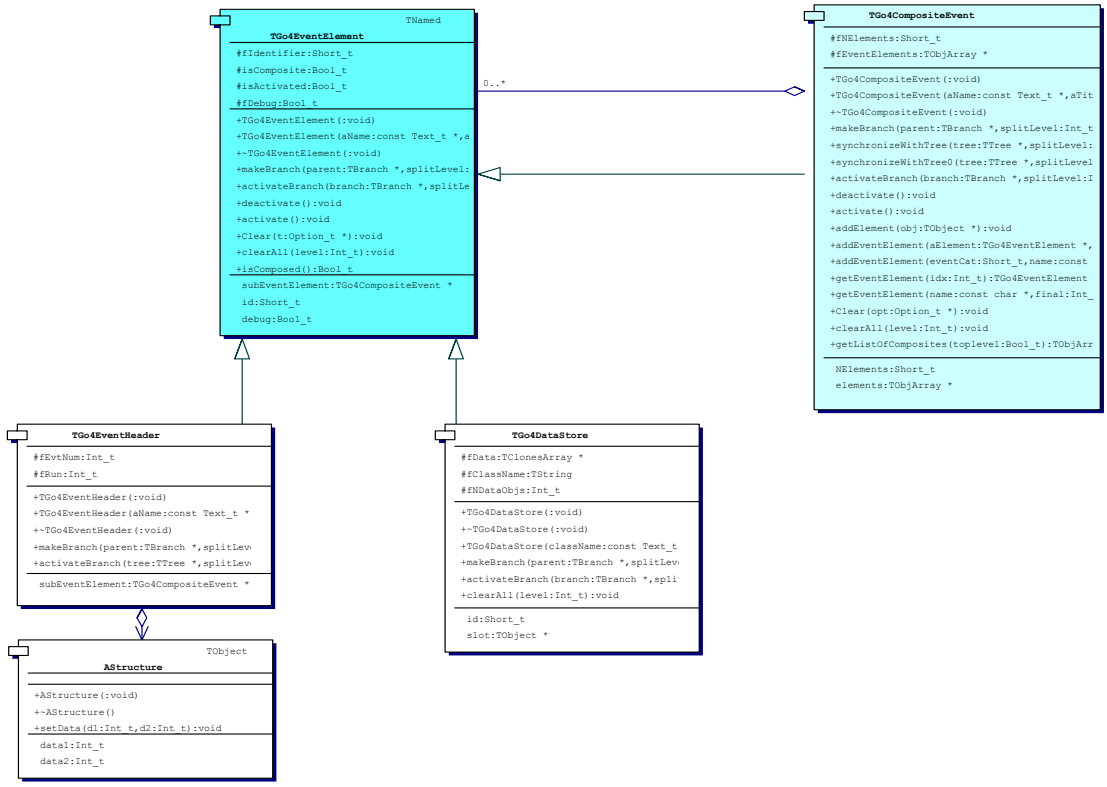
The design of the event object should be general enough in order to be adapted by different experiments, should not create performance penalty compare to native ROOT tree structure when streaming the objects in file and should provide an easy-to-use user interface hiding the underlying ROOT internal data representation.

9.2 *Implementation*

According to the general description, it is clear that an event should be defined as a composite object. To implement such a composite object, the *composite design pattern* has been used. The *composite design pattern* is an abstraction model that allows a client to treat individual objects and composition of objects uniformly. It allows to build complex objects by recursively composing similar objects in a tree-like manner. It also allows the objects in the tree to be manipulated in a consistent manner, by requiring all of the objects in the tree to have a common super-class or interface.

Using such a general design, one can build complex structure out of simpler one. And then build even more complex things based on what you have already build and so on ... But you would like to treat all of them the same way.

The following picture shows the UML diagram of the composite event.



The core classes are shown in color. An abstract component ***TGo4EventElement*** declares the interface for object to be composed. It also implements default behavior common to all classes.

An abstract composite class ***TGo4CompositeEvent*** is defined as a subclass of ***TGo4EventElement*** interface and has no specific behavior itself. The ***TGo4CompositeEvent*** acts as a container class of classes inherited from ***TGo4EventElement*** and as it is itself deriving from the same abstract interface, it could contains also classes inherited from ***TGo4CompositeEvent*** . Therefore one as no limitation in composing complex objects in a tree-like structure. Furthermore, this abstract composite class is the natural place for the implementation of pure recursive algorithms. For instance, when the function *TGo4CompositeEvent::getEventElement(const char* name)* is invoked, it is simply sequentially sent to all its children in order to find the one matching the name. The composite event class implements child-related operations.

The user can add its own components for modeling its own event structure by sub-classing either directly the ***TGo4EventElement*** class or the ***TGo4CompositeEvent*** class. The diagram above shows 2 different user-defined subclasses i.e. ***TGo4EventHeader*** and ***TGo4DataStore*** which both contains specific data-members or data-containers that has to be streamed in a binary ROOT file format. Go4 provides an equivalent of ***TGo4DataStore***: ***TGo4ClonesElement***.

9.3 *User interface*

The process of modeling an event structure will now be described. In order to benefit from the composite model functionality, one will have to create either sub-classes of ***TGo4EventElement*** or sub-classes of ***TGo4CompositeEvent*** according to the following scheme:

- Elementary data objects inherit from ***TGo4EventElement*** . These objects are elementary bricks of the data structure which contain members of all data-types that the ROOT system supports in its IO split mechanism.

```
{ enum (kSize=10);
  Char_t fType[20];       // array of characters
  Int_t fNTracks;         // single type int
  Int_t  fX[kSize];       // an array where dimension is an enum
  Float_t fMatrix[4][5];  // 2- dimensional array of floats
  Float_t *fDistance;     //[fNTracks];
  TString fNames[12];     // array of TString
  EventHeader fEvtHdr;    // example of class not derived from TObject
  TClonesArray *fData;    // array of cloned TObjects
  TH1F* h1;               //-> pointer to histogramms
  TArrayF fArrayF;        // an array of floats
  TArrayI *fArrayI;       // a pointer to an array of integer
}
```

All these example data-members can be used when sub-classing the base class ***TGo4EventElement*** . The user-defined class will then be translated into a ***TBranchElement*** object that represents the basic element of the ROOT ***TTree*** object. The corresponding ***TBranchElement*** in turn will contain a list of ***TLeaf*** objects for every data member appearing in the header class definition.

```
Class MyDerivedClass: public TGo4EventElement{

// private data menbers
   Int_t data[10][20];
   MySpecificObject  fMyobj; // specific data-object should not derived from Tobject
// etc ...

   MyDerivedClass();
   MyDerivedClass(const Text_t *aName,const Text_t *aTitle,Short_t aBaseCat);
  ~MyDerivedClass();
// some public user-defined function for this class
// etc ...

  }
```

In order to create an elementary object, one should follow the ***TGo4EventElement*** general interface, and according to this interface the data object should have as parameter of its constructor

- a name
- a title
- a unique identifier

The name will be used to generate the corresponding ***TBranchElement*** branch names in the ROOT ***TTree*** layout. The identifier should be **unique** for each user class. It is good style to define all identifier in a global header file in the project as follow:

```
// file MyIndexesDefinition.h
{
#ifndef __ElementIndexes_H
#define __ElementIndexes_H

// Fast Bus nodes position in composite structure
 const Int_t fastbusID=0; // defined reserved area of index for fastbus object i.e [0,10]
 const Int_t Ifastbus1=fastbusID+1;
 const Int_t Ifastbus2=fastBusID+2;
 const Int_t Ifastbus3=fastBusID+3;
// etc …

// slot nodes position in composite structure
    const  Int_t slotID=10 ; // defined reserved area of index for slot objects i.e [10,40]
     const Int_t Islot1 =  slotID+1 ;
    …

#endif
}
```

One can also define data-container class that inherits from ***TGo4CompositeEvent.*** i.e.

```
// file MyCompositeEvent.h
#ifndef MyCompositeEvent_H
#define MyCompositeEvent_H
#include "Go4Event/TGo4CompositeEvent.h"
#include "Go4Event/TGo4ClonesElement.h"

#include "FB.h"
#include "CAM.h"
#include "VME.h"
#include "MyIndexesDefinition.h"

class MyCompositeEvent : public TGo4CompositeEvent
   {
 public:
     Int_t EvNumber;              // just event number
     Int_t FBNumber;              // number of FB data for event# EvNumber
     Int_t CAMNumber;             // number of CAM data for event# EvNumber
     Int_t VMENumber;             // number of VME data for event# EvNumber

 public:
   MyCompositeEvent();
   MyCompositeEvent(const char* name, const char* title,int index);
  ~MyCompositeEvent();
  void SetT4pEventHeader(Int_t inumber){EvNumber = inumber;}
  Int_t GetEvNumber() {return EvNumber;}
```

```
  void AddF(Int_t FNumber, Int_t Slot, Int_t Chan, Int_t Data);
  Int_t Fill();

 ClassDef(MyCompositeEvent,1)
};
#endif // !MYCOMPOSITEEVENT_H
```

The class ***MyCompositeEvent*** will create the complex structure of the user-defined event inside its constructor i.e:

// file MyCompositeEvent.cxx

```
MyCompositeEvent::MyCompositeEvent(const char* name, const char* title):
    TGo4CompositeEvent(name,title)
    {
   // create a Composite structure for each FastBus
     TGo4CompositeEvent *fb1 = new TGo4CompositeEvent("FB1","FastBus1",Ifastbus1);
     TGo4CompositeEvent *fb2 = new TGo4CompositeEvent("FB2","FastBus2",Ifastbus2);
     TGo4CompositeEvent *fb3 = new TGo4CompositeEvent("FB3","FastBus3",Ifastbus3);

    MyDerivedClass *data1,*data2,*data3;
    data1 = new MyDerivedClass("DataFB1","TDataFB1",Islot1);
    data2 = new MyderivedClass("DataFB2","TDataFB3",Islot2);
    data3 = new MyDerivedClass("DataFB3","TDataFB3",Islot3);
   // add data object to each fastbus composite event
   fb1->addEventElement( data1 );
   fb2->addEventElement( data2 );
   fb3->addEventElement( data3 );

   // add One DataObject to each fastbus node in MyCompositeEvent
   addEventElement( fb1 );
   addEventElement( fb2 );
   addEventElement( fb3 );
    }
```

This example shows how to create an event structure out of elementary user-defined data-objects class i.e. ***MyDerived-Class*** simply using the function :
*TGo4CompositeEvent::addEventElement(TGo4EventElement * evt);*
The structure can then be browsed from the Go4 tree viewer.

10 Icon Table


File pad: open local ROOT file on disk


File pad: open remote ROOT file (TNetFile, TWebFile, TRFIOFile)


Save content of memory to ROOT file


File pad: close selected ROOT file


File pad: close all ROOT files


Export selected objects of memory browser into another format (ASCII, radware, ROOT)


Stop running analysis, shutdown analysis and terminate GUI


Open view panel


Open fitter window


Open histogram properties window (there: list properties in analysis window)


Open histogram creator window


Open condition properties window (there: list properties in analysis window)


Open condition editor


Open event inspector window


Open dynamic list editor


List dynamic list in analysis window


Open parameter editor


Open browser to (un)load libraries; show list of loaded libraries


Open user GUI


Open analysis launch window


Stop and shut down analysis client, disconnect analysis server


Stop and shut down analysis server


Start analysis. Monitor pad: start updating all objects in list, or only displayed ones.


Stop analysis


Open analysis configuration window (can be closed/opened any time); browser popup menu: edit selected


Open analysis output window (can be closed/opened any time)


Open file browser


Open color editor


Expand/shrink histogram in selected pad in X.


Expand/shrink histogram in selected pad in Y


Expand/shrink histogram in selected pad in Z.


Move expanded histogram in selected pad in X direction


Move expanded histogram in selected pad in Y direction

Move expanded histogram in selected in pad Z direction


Set fill color


Set line color


Set marker color


Scale Y axis linear/logarithmic


Scale Z axis linear/logarithmic


Scale X axis linear/logarithmic

Draw 1d histogram/line style

Reset display in selected pad to histogram limits

Open window to set display limits (applies to selected pad, or all pads if this option is enabled in view panel)

Execute Tree draw.

Kill analysis

Clear button in browser pads clears objects in analysis, in condition editor clears counters.

Enable clear function for objects

Disable clear function for objects (does not clear these objects)

Analysis pad: copy selected object(s) to monitor

Remove selected object(s)

Move selected object(s) to memory (from analysis, monitor, or histogram server); or copy object from analysis to editors (conditions or parameters)

Copy object in editor to analysis (conditions or parameters)

Analysis pad: update folders from analysis. Memory pad: update all objects from analysis and redraw.

Browser icons for window condition (arrays). Window mode in marker editor

Browser icons for polygon condition (arrays). Polygon mode in marker editor

Browser icon for TCanvas

Browser icon for TGraph

Browser icon for Go4 pictures.


Brower icon for TH3 histograms


Browser icon for TH2 histograms


Browser icon for TH1 histograms. Button: draw selected objects (one per pad).


Draw selected objects (all in one pad, superimpose)


Save selected object in memory to ROOT file


Refresh memory list (needed to see new histograms created e.g. by ROOT in the GUI). In condition editor: refresh values from view panel.


Browser popup menu: open information window for selected histogram or condition


Editors: shows up if object in editor differs from object in analysis (file). Use  for update.


Condition editor: connect to a picture with conditions (gets list of conditions from it)


Condition editor: update graphics from values in editor.


Output values of condition editor, info window, or markers according settings in the log action.


Close window without further action


Browser icon for dynamic list entries


Insert arrow in marker editor


Pick next mouse click in pad to get values into condition editor or marker editor

Browser icon for a tree


Browser icon for a branch


Browser icon for leafs

11 Table of Menu Keyboard Shortcuts

Note that the **Alt-x** keys work on windows whereas the **Ctrl (Strg)-x** keys work directly. Sometimes the same function is available in both, i.e. **Alt-a-n** or **Strg-n**. In these cases the last character is identical.

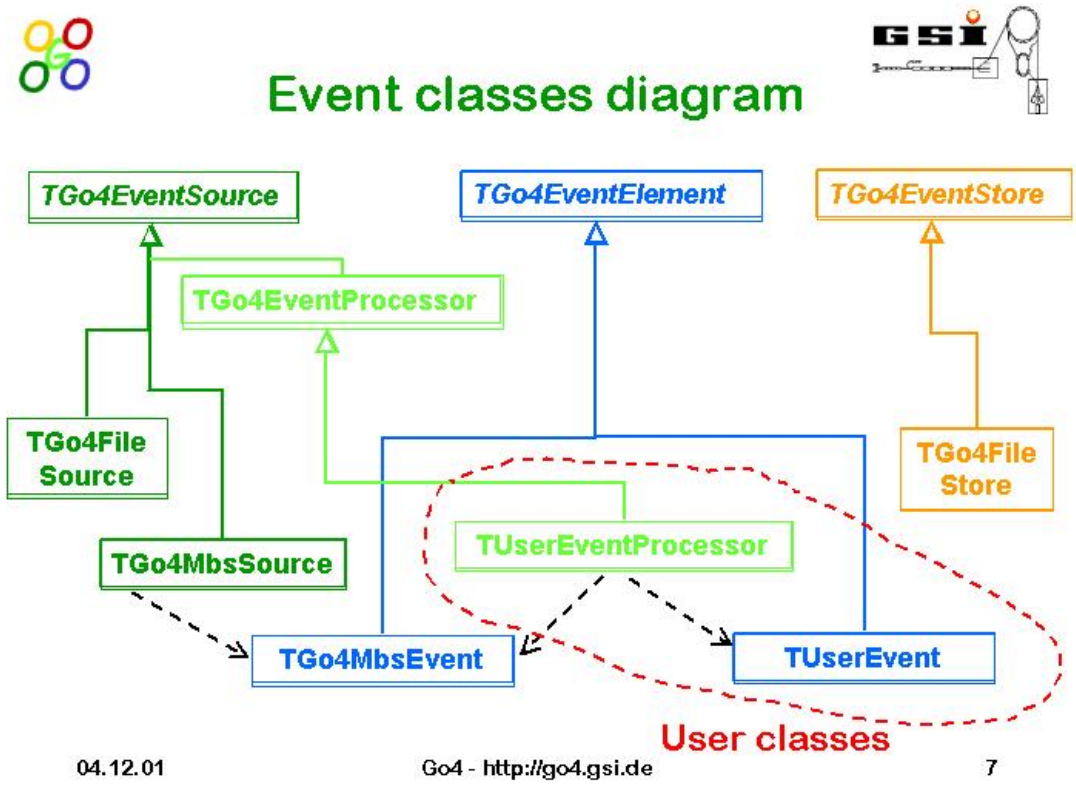
Ctrl-O	**Alt-F-O**	File menu: **O**pen local file
Ctrl-R	**Alt-F-R**	File menu: Open **R**emote file (TNetFile, TWebFile, TRFIOFile)
Ctrl-Y	**Alt-F-Y**	File menu: Save all objects of memor**Y** browser to ROOT file
Ctrl-Q	**Alt-F-Q**	File menu: Close (**Q**uit) all files
Ctrl-X	**Alt-F-X**	File menu: E**X**it Go4
Ctrl-V	**Alt-T-V**	Tools menu: Open new **V**iew panel
Ctrl-F	**Alt-T-F**	Tools menu: **F**itpanel
-	**Alt-T-H**	Tools menu: **H**istogram properties window
Ctrl-I	**Alt-T-I**	Tools menu: H**I**stogram creation tool
-	**Alt-T-O**	Tools menu: C**O**ndition properties window
Ctrl-C	**Alt-T-C**	Tools menu: General **C**ondition editor
-	**Alt-T-E**	Tools menu: **E**vent printout and inspection tool
Ctrl-D	**Alt-T-D**	Tools menu: General **D**ynamic list editor
Ctrl-B	**Alt-T-B**	Tools menu: Load li**B**rary dialog
Ctrl-U	**Alt-T-U**	Tools menu: **U**ser GUI
Ctrl-N	**Alt-A-N**	Analysis menu: Lau**N**ch analysis client process
Ctrl-M	**Alt-A-M**	Analysis menu: Disconnect (re**M**ove) analysis client
Ctrl-T	**Alt-A-T**	Analysis menu: Submi**T** settings and start analysis run
Ctrl-S	**Alt-A-S**	Analysis menu: **S**tart analysis run
Ctrl-H	**Alt-A-H**	Analysis menu: Stop (**H**alt) analysis run
Ctrl-G	**Alt-A-G**	Analysis menu: Show/hide analysis confi**G**uration window
Ctrl-W	**Alt-A-W**	Analysis menu: Show/hide analysis output terminal **W**indow
-	**Alt-S-O**	Settings menu: sh**O**w/hide...
-	**Alt-S-F**	Settings menu: **F**onts...
-	**Alt-S-Y**	Settings menu: St**Y**les...
-	**Alt-S-L**	Settings menu: **L**og actions
-	**Alt-S-H**	Settings menu: Generate **H**otstart
-	**Alt-S-T**	Settings menu: Analysis **T**erminal history length
-	**Alt-S-S**	Settings menu: **S**ave Settings
-	**Alt-W-S**	Windows menu: Ca**S**cade
-	**Alt-W-T**	Windows menu: **T**ile
-	**Alt-W-C**	Windows menu: **C**lose all windows
-	**Alt-W-M**	Windows menu: **M**inimize all
-	**Alt-W-O**	Windows menu: Save L**O**g window to text file
-	**Alt-W-L**	Windows menu: Clear **L**og window
-	**Alt-W-A**	Windows menu: Save **A**nalysis window to text file
-	**Alt-W-W**	Windows menu: Clear analysis **W**indow
-	**Alt-H-I**	Help menu: Read Go4 **I**ntroduction manual
-	**Alt-H-R**	Help menu: Read Go4 framework **R**eference manual
-	**Alt-H-F**	Help menu: Read Go4 **F**itpackage manual
	Alt-H-G	Help menu: Read Go4 **G**UI macro command reference
F2	**Alt-H-Q**	Help menu: About **Q**t
F3	**Alt-H-O**	Help menu: About R**O**OT
F4	**Alt-H-G**	Help menu: About **G**o4
Alt-U	**-**	If analysis configuration window is active: S**U**bmit analysis settings

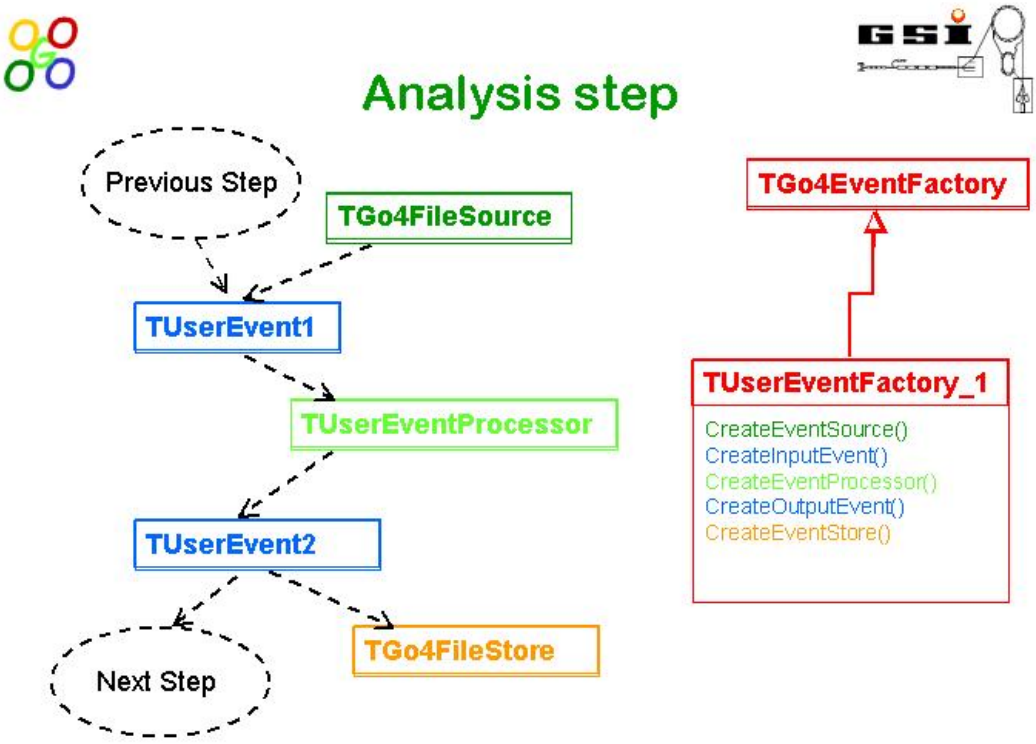
-	**Alt-I-S**	View panel file menu: **S**ave as...
-	**Alt-I-P**	View panel file menu: **P**rint...
-	**Alt-I-O**	View panel file menu: Cl**O**se View panel
-	**Alt-E-E**	View panel edit menu: Show/hide marker **E**ditor
-	**Alt-E-R**	View panel edit menu: Show/hide **R**OOT attributes editor (TGedEditor)
-	**Alt-E-C**	View panel edit menu: Start **C**ondition editor and work on pad conditions (in pictures)
	Alt-E-E	View panel edit menu: Show/hide object **E**vent status line
-	**Alt-E-1**	View panel edit menu: Change to **1**:1 coordinates ratio
-	**Alt-E-D**	View panel edit menu: Change to **D**efault pad margins
-	**Alt-E-M**	View panel edit menu: Clear **M**arkers
-	**Alt-E-P**	View panel edit menu: Clear **P**ad
-	**Alt-E-A**	View panel edit menu: Clear C**A**nvas
-		
-	**Alt-O-C**	View panel options menu: Toggle **C**rosshair mode
-	**Alt-O-S**	View panel options menu: Show/hide histogram **S**tatistics box
-	**Alt-O-T**	View panel options menu: Show/hide histogram **T**itle box
-	**Alt-O-L**	View panel options menu: Show/hide multiplot **L**egend
-	**Alt-O-K**	View panel options menu: **K**eep view panel title
-	**Alt-O-V**	View panel options menu: Set **V**iew panel title...
-	**Alt-O-I**	View panel options menu: Toggle Super**I**mpose mode

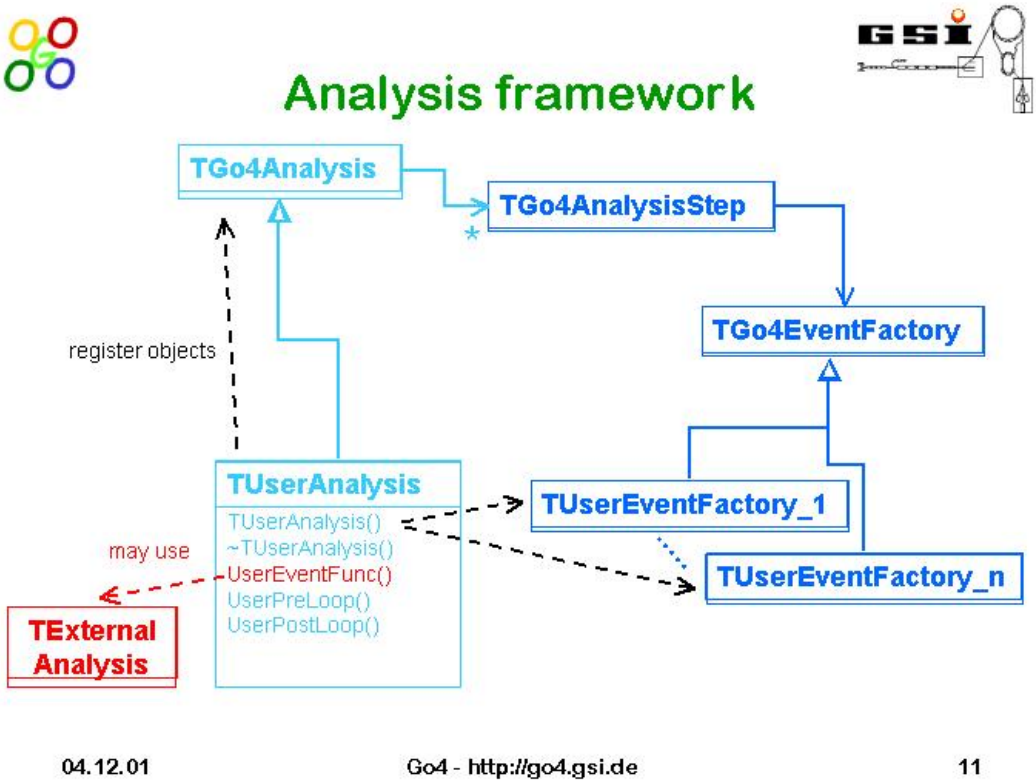
12 Event Classes Diagrams

The following UML scheme gives an overview of the event base classes and typical implementations:

The **TGo4MbsEvent** is filled from the **TGo4MbsSource** (both provided by Go4). The **TUserEventProcessor**, which had been defined to match the user's experiment, takes the raw data from GSI format 10,1 and unpacks them into the **TUserEvent** object. Both **TGo4MbsEvent** and **TUserEvent** objects can be stored into (different) **TGo4FileStore** instances. Later these can be read again event-by-event using the **TGo4FileSource**.




Analysis step
Previous Step
TGo4FileSource
TUserEvent1
TUserEventProcessor
TUserEvent2
TGo4FileStore
Next Step
TGo4EventFactory
TUserEventFactory_1
CreateEventSource()
CreateInputEvent()
CreateEventProcessor()
CreateOutputEvent()
CreateEventStore()
04.12.01
Go4 - http://go4.gsi.de
10


Analysis framework
TGo4Analysis
TGo4AnalysisStep
*
TGo4EventFactory
register objects
TUserAnalysis
TUserAnalysis()
~TUserAnalysis()
UserEventFunc()
UserPreLoop()
UserPostLoop()
TUserEventFactory_1
TUserEventFactory_n
may use
TExternal Analysis
04.12.01
Go4 - http://go4.gsi.de
11

13 Index