Eurogam project **EDOC119**

Developpement graphique

DISPLAY MODULE - ANALYSIS MODULES

Edition 1.0
20 February 1991

G.ZEHNACKER et L. MICHEL

Centre de Recherches Nucléaires de Strasbourg
(SATD)
CNRS-IN2P3 France

Abstract

The users of the future softwares for data analysis need to use her own treatment modules. We must then imagine a display kernel which uses functions coming from different external libraries. The interface between the external libraries and the kernel don't must need modification of one or the other.

We propose in this document a method to do that, using a X toolkit for the kernel and executable files for the user's functions, the both communicating trough the UNIX shell.

1 Display module - analysis modules.

Some applications must be able to display spectra and to analyse them. The software used for display and manage images must be very secure. We think that it must be written like a graphic kernel, tested in detail, and not modifiable. This kernel will be used in some different applications for the data analysis and the experience control. This way, the method for manipulate images will be known by each user, and well-known for its reliability.

The application doesn't only manipulate images but also do calculations on data. These other functions are specific to each application (and user ?) and then, must be easly linkable to the graphic kernel. They must be callable from other contexts that our display software and then they must be stored in external librairies.

2 Using the shell

The external functions must have a specific format to be used in different contexts. The most simple way to call functions is to use the shell on an alphanumeric terminal. In this case, the functions must be executable files using the standard input and output. Her callback from the shell is the better way to debug them and to write small applications.

The Unix system has a lot of abilities to interface the shell and the applications. So we think that the external functions must be callable from the shell, and from our applications trought the shell.

Then they will be callable without specific adaptation by all the applications runing under Unix.

3 Calling external functions

3.1 The five steps to call external functions

The call by an application of an external function has five steps needing each a specific treatment (see Fig 1):

- Select the function.
- Set the paramters of the function.

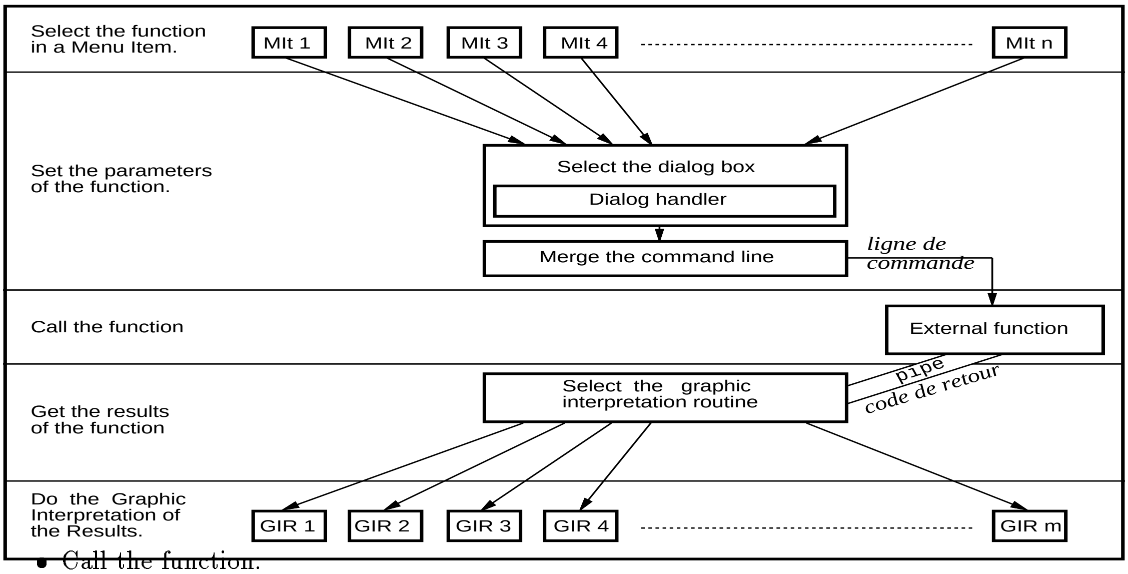

Fig 1: Calling an external function.

- Call the function.
- Get the results of the function.
- Do the graphic interpretation of the results.

3.2 Selecting the function

The external functions are called from menu items. The contenu of a menu and its label are set up specifically for the external functions library.

Four menus (help, storage, image and data-base) are managed directly by the graphic kernel. The other menus call external functions.

3.3 Setting the paramters

Each external function has specific arguments, with a specific format, in a specific order.

The calling application must know this format and then sets from the user requsts the corresponding command line. Our idea is to do that with dialog boxes having an active object associated with each argument (for more details, see section 4.2).

3.4 Calling the function

The call of the functions must use characters strings sent to the shell. The process must have its output redirected to the calling program. That can be done by a Unix pipe (see example section 3.5).

3.5 Getting the results

The called function is connected to the calling programm by a Unix pipe. So the results can be read by the calling programm on the stream associated to the pipe.

Example:

```
char      *command_line = ”function_name arg1 arg2 arg3”;
FILE      *pipe, *popen();
char      result_string[RESULT_SIZE];
          .
          .
pipe = popen(command_line , ”r”);
while( fgets(result_string,RESULT_SIZE,pipe) ) parse_result(result_string);
          .
          .
```

The command line is merged in the string **command_line* from the name of the function (selected in a menu) and the parameters set in a dialog box. This command is sent to the shell by the callback *popen* , which starts up the function with its output connected on the stream *pipe* . That calling programm reads the result strings on this stream with the function *fgets* . The results are sent to the appropriate graphic routine which is selected by the *parse_result* function (for more details, see subsection ...).

3.6 Doing the graphic interpretation

Some external functions return single values which must be just displayed in characters what doesn't set problem. But some other functions return complex results such as spectra, peaks descriptors or peaks fits. In this case, the calling program (graphic kernel) must be able to interpret graphically these values. It must draw the spectra, the peaks or the fits from the returned values.

The problem is that the user's functions (external) doesn't describe the same objects with the same formats. In consequence, the graphic kernel must have a library of functions able to interpret the results in the currentest formats. If a user want to add a new function which returns data with a new format, then a specific function will must be added to the graphical kernel to interpret the results.

When an user sets up his graphic kernel with his library, he must specify for each external function the routine which must be called to interpret the results (for more details, see subsection 4.4).

4 Structure of the graphic kernel

4.1 A parametrable kernel

The manipulation of the image, the access to the file system and the graphic interpretation are done by the graphic kernel. The same graphic kernel must be used with different libraries of external functions without modification.

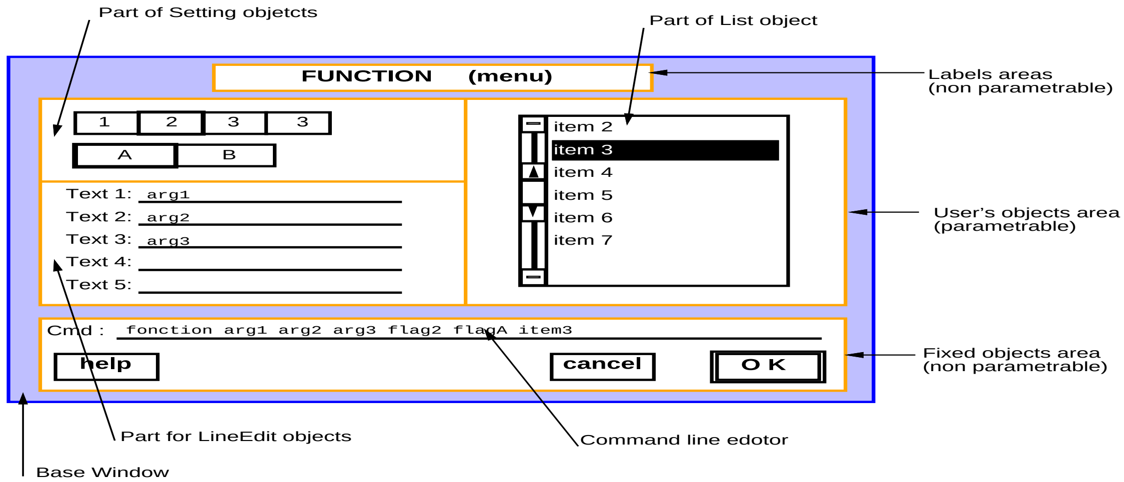

Fig 2: General structure of a dialog boxe.

That can be done if the graphic kernel is set up at its start up. This setup defines the contenu of the menu bar and the menu items, the features of the associated dialog boxes and the associated routine for the graphic interpretation of the results.

All these informations can be written in an ASCII file associated to each library. In the following subsections we will define the most important features of these files.

4.2 Setting the dialog boxes

Each call to an external function is parametred with a dialog box handler, which read the arguments in the box and merges the command line.

The features of the dialog boxe depend on the external functions, they must be set up from the configuration file. Our idea is to authorize the use of a limited set of active objects (widget such as setting, line edit and list) and to associate a single argument to each one. The look of the dialog boxe is realized by the graphic kernel from the list of active objects. The setup file gives for each dialog boxe some general parameters (movable, resizable, focused,..) and a list of objects with their own parameters.

The graphic kernel reads this file when it starts up, and then creates the set of boxes.

All the boxes have some common objects which doesn't be defined in the setup file. This objects are three buttons (OK, Cancel and Help) and an line editor in which is written the command line.

We suggest to set three areas (see Fig 2) in a dialog box (each area can correspond to a XView control area, if XView is used). In the area on the top of the boxe is written the name of the command, and its menu. In the area of the bottom of the boxe are set the common objects (buttons, command line). In the middle of the boxe are set the user's dependents objects.

The user can set just three classes of object: the setting, the line edit and the list. Each class has a reserved part of the middle area. The size of these parts depends on the number of items inside. If it is no object, the part has a null size.

The size of the boxe (see Fig 3)depends on the number of objects. To create the boxe, the graphic kernel calculates the size of each part, next, the size of the central area and

Complete configuration

Setting + line edit

Line edit + scrolling list

Setting + scrolling list

Command line alone

Fig 3: different configurations of the dialog boxe

then, pastes the three areas on a base window.

Each dialog boxe is described in memory by a structure which type is not yet defined (see subsection ...). So a unique routine can handle all the dialog boxes and return their command lines.

4.3 Setting the menu bar

The menu bar is empty, except four buttons, when the graphic kernel starts. Four menus are managed by the kernel: the File menu, which permits the acces to the file systems, the Image menu, which permits the manipulation of the images and the help and the data-base menu. Some internal functions can be called from command buttons without use a menu (see EDOC 112).

The other menus, empty when the applicvation starts, are filled in from the setup file.

At the begining of this file, the label of each menu is written. The item label, and its menu name is specified at the begining of the definition of each dialog boxe.

The menu bar is set up during the analysis of the set up file.

4.4 Interpreting the results

We have seen (subsection 3.6) that the graphic kernel has some routines which interpret graphically specific result formats. All the results are passed to these routine by the string read on the pipe (see subsection 3.5). The problem is then to select the appropriate routine from the definition of the external callback written in the setup file.

The graphic interpretation routine can be indentified by an integer or a string in the setup file. That value belongs to the structure defining the callback and is used by the *parse_result* function (see subsection 3.5) to switch to the good routine. Its also possible to set a pointer in that structure pointing to the interpretation routine (see following example).

Example:

```
typedef struct { Dialog       dialog;
                 Menu         menu;
                 int          *interp_fct(); } Callback_def;
Callback_Def     callback_def;
.
.
init_callback(callback_def); /* init the callback_def struct */
                 .
extern_callback(Callback_Def callback_def) {
char             *command_line[MAX_LGTH_CMD];
FILE             *pipe, *popen();
char             result_string[RESULT_SIZE];

dialog_handle(callback_def, command_line);
pipe = popen(command_line , ”r”);
while( fgets(result_string,RESULT_SIZE,pipe) ) callback_def.interp_fct(result_string);
                 .
                 .
}
.
.
```

Each external callback is described by a structure of type *Callback_Def* which contains the definition of the dialog box, the reference of the menu and a pointer to the graphic interpretation function. When the function is selected from its menu, the routine *extern_callback* is called to manage the callback. It first calls the dialog handler *dialog_handle* which merge the command line in the string *command_line* . After read the result string the graphic interpretation routine *callback_def.interp_fct* pointed in the *callback_def* is called.

4.5 The setup file

We have now enough elements for define the most important features of the setup file. The external callbacks have three levels of definition:

- *The menus:* Which specify the menu labels and the menu items.
- *The dialog boxes.* Which specifies the graphic options, the graphic interpretation routine and the structure of the command line.
- *The graphic objects.* Which specifies for each object the graphic options and the associate argument,

Example of syntax for the setup file:

```
###############################
#          SETUP FILE
#
#    SYNTAX PROTOTYPE
###############################

###############################
# definition of the first menu.
###############################
MENU LabelMenu1 Pinable Movable
    LabelItem1 LabelItem2 . . . LabelItemN

###############################
# definition of the 2nd menu.
###############################
    .
    .
    .
    .

###############################
# definition of the Mth menu.
###############################
MENU LabelMenuM NoPinable Fixed
    LabelItem1 LabelItem2 . . LabelItemK

###############################
# definition of the 1st dialog box
###############################
DIALOG LabelMenu1:LabelItem1 Resizable Movable Focusable NonEditabelCommandLine
    OBJECT_SETTING LabelSetting1 ExclusiveChoise
        LabelItem1 LabelItem2 ... LabelItemN
    OBJECT_SETTING LabelSetting2 NonExclusiveChoise default=OldValue
        LabelItem1 LabelItem2
    OBJECT_LINEDIT LabelLinedit1 NumericOnly ChanelClick
    OBJECT_LINEDIT LabelLinedit2 NumericOnly ChanelClick
    OBJECT_LINEDIT LabelLinedit3 AlphaNumeric Default="-1"
    OBJECT_LIST LabelList1 ReadOnly ExclusiveChoise
        Item1 Item2 ... ItemN
    GRAPH_INTER_ROUTINE DrawPeakFromGaussienne
    COMMAND_LINE LabelLinedit1 LabelLinedit2 LabelSetting1 LabelSetting2 LabelList1 LabelLinedit3

###############################
# definition of the 2nd dialog box
```

```
################################
DIALOG LabelMenu1:LabelItem2
     .
     .
     .
     .
```

The configuartion file has one definition for one object per line. If a definition line is too long for the text editor it's possible to cut it in several parts with a at the end of each one. The structure of a definition line is :

> *[type of object]<blank>[label of object]<blank>[options li*
> *<tabulation>[items list]*

Each field can be set by a keyword. The following list gives just some examples, the global set of keywords will be defined later:

[type of object] Keyword giving the class of the defined object:

MENU : Definition of the items and the label of a menu.

DIALOG : General definition of a dialog boxe.

OBJECT_SETTING : Definition of a setting included in a dialog boxe.

OBJECT_LINEDIT : Definition of a linedit included in a dialog boxe.

OBJECT_LIST : Definition of a list included in a dialog boxe.

GRAPH_INTER_ROUTINE : Definition of the internal routine used for the graphic interpretation.

COMMAND_LINE : Order of the arguments in the command line.

[Label of object] The label can be a string (between "") or a menu label (menulabel:itemlabel:...)

[options list] Keywords, separeted by a blank, defining the mode of use of the defined object:

Pinable (NoPinable) : The menu or the dialog can be pined on the screen.

Movable (Fixed) : The menu or the dialog can be moved on the screen.

Focused (Unfocused): The events are focused on the dialog.

Resizable : The dailog can be resized.

EditableCmdLine : The user can write directly the commandline.

Editable (ReadOnly): The object can be set by the user.

ExculsiveChoise : The list or the setting can select a single item.

Default=value: Definition of the default value for an object Value can.

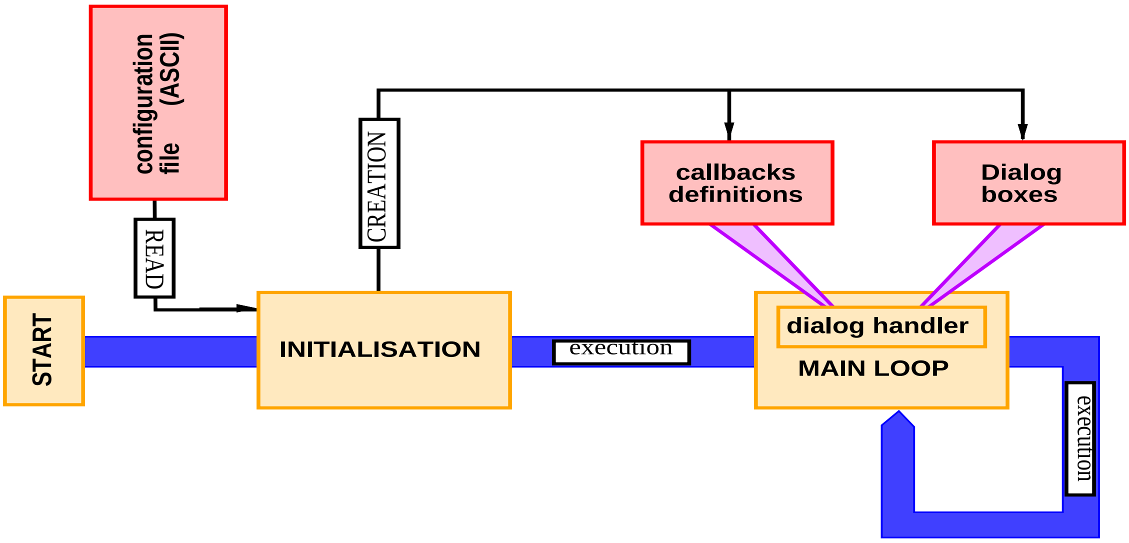

Fig 4: Creation and use of the data defining the callback.

be a string (between ""), a number for a setting or a keyword (OldValue).
Default=OldValue: the default value is the est value of the last call)
. **NumericOnly :** The linedit or the list can be set just with numeric values.
AlphaNumeric : The linedit or the list can be set with all characters.
ChannelClick : The line edit or the list can be set with a channel number selected with the mouse in a draw of a spectrum.

[items list] Used for the objects having several items (list, setting, menu). The items list is written on the line under the object definition. Its line gegin with a tabulation character.

At the begining of the file are defined the menus. After, all the dialog boxes associated with each menu item are described.

All the words are user's defined strings if written between "", or keywords known by the graphic kernel.

4.6 Structure of a callback in memory

Data structures describing the interface between the kernel and the external functions are created by the module of initialisation of the graphic kernel during its initialisation (see Fig 4).

Each class of active object is defined by a specific structure type (see the following code lines). This structure gives the number of objects, a list of pointers on these objects, the place of each associated argument in the command line and the of options. The list of pointers and positions ever can contain the maximum number of items (MAX_XXX). That takes up uselessly a little bit of memory, but not so much to justify the use of dynamic allocation.

```
LIST STRUCTURE: typedef struct { int      nb_obj;
                                 XSetting *ptr_obj[MAX_SETTING];
                                 Option   option_list;
                                 int      posit[MAX_SETTING];} Setting;

                typedef struct { int      nb_obj;
                                 XLinedit *ptr_obj[MAX_LINEDIT];
                                 Option   option_list;
                                 int      posit[MAX_LINEDIT];} Linedit;

                typedef struct { int      nb_obj;
                                 XList    *ptr_obj[MAX_LIST];
                                 Option   option_list;
                                 int      posit[MAX_LIST];} List;
```

A callback is defined by a structure type (see the following code lines) which contains the definition of the active objects (see above), the pointer to the menu item at which the callback is connected and the pointer to the routine of graphic interpretation of the results.

The use of the routine of graphic interpretation is explained in the subsection 4.4.

```
CALLBACK STRUCTURE: #define        MAX_OBJ MAX_SETTING + MAX_LINEDIT + MAX_LIST

                    typedef struct { Setting setting;
                                     Linedit linedit;
                                     List    list;
                                     XMenu   menuitem;
                                     XObj    cmd_line[MAX_OBJ];
                                     int     *interp_fct(); } Callback_Def;
```

We discuss here only on the principle of the data representations. The details of the structures will be explain later. Notably, the structure of the options list and XObjects types, which depend on the used X toolkit.

5 Conclusion

We have described in this document how making a user's configurable analysis appliaction using the concepts of the currentest XToolkits (the widgets).

We think that the user's functions doesn't call graphic routines. For this reason, we put all the graphic functions in our graphic kernel. This way, the user's functions are callable in all contexts and the reliability of the graphic management is assumed.

The interface between our kernel and the user's functions is realized with an ASCII file easy to read and to write. The look of the dialog boxes is automatically created by the graphic kernel, it doesn't figure in the configuration file.

It seems interesting to us to integrate the method developped in this document in other applications that the data analyser.