SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2002; **32**:873–887 (DOI: 10.1002/spe.465)

Command-line interfaces can be efficiently brought to graphics: COLIMATE (the COmmand LIne MATE)

C. O. S. Sorzano[1,*,†], J. M. Carazo[1] and O. Trelles[2]

[1]*National Center for Biotechnology, Campus Universita Autónoma s/n, 28049, Cantoblanco, Madrid, Spain*
[2]*Departamento Computer Architecture, Complejo Politécnico, Campus de Teatinos, Apartemendo 4114, 29080, Málaga, Spain*

SUMMARY

Scientific computing has traditionally been done on workstations, most of the time running on UNIX-like operating systems. These systems are often very robust, powerful and especially suited to heavy computation tasks; however, the usual user interface is based on a command line with all the information needed for the appropriate functioning of the algorithm. These command lines are seldom user-friendly but, on the other hand, they allow batch processes to be built. In this work, a model for command-line-driven packages is given, and at the same time the model includes objects that can be directly translated into a graphical user interface. A language (Colimate) implementing this model is shown to serve as a bridge between command-line-driven programs and more friendly user interfaces. Exploiting the specificness of the user interfaces addressed, powerful graphical interfaces can be built with a very simple syntax and small programming effort. Furthermore, the command-line program needs not to be modified, so in this way the possibility of making batches is not lost. Copyright © 2002 John Wiley & Sons, Ltd.

KEY WORDS: user interface; GUI; command line; rapid development

INTRODUCTION

The worth of graphical versus textual interfaces has been extensively proved, particularly for those non-advanced computer users who are interested only in the package functionality. The traditional use

*Correspondence to: C. O. S. Sorzano, National Center for Biotechnology, Campus Universita Autónoma s/n, 28049, Cantoblanco, Madrid, Spain.
†E-mail: coss@cnb.uam.es

Contract/grant sponsor: Comisión Interministerial de Ciencia Y Tecnología; contract/grant number: BIO98-0761
Contract/grant sponsor: EU-FEDER; contract/grant number: 1FD97-0372

Received 7 November 2001
Revised 25 February 2002
Accepted 2 March 2002

of text interfaces for programs became obsolete in the Macintosh and Windows platforms much earlier than in the Unix environment where it is still alive, for instance, in the scientific community [1–3]. This fact can be explained in part by several reasons: the user population addressed by the scientific programs is much smaller than the general public usually addressed by other software fields; scientific packages are often developed centering the efforts on the software functionality and not on the user interface; the absence of visual languages and the relatively late appearance of easy use graphical libraries [4–12] in Unix-like platforms, the common absence of commercial interests in scientific software who could afford the extra time and cost of the development of graphical interfaces, etc.

As a result, non-commercial scientific applications developed in Unix environments usually show a simple text interface, most of the time as a command line. The main features of a command line can be summarized in two points: they are very simple to program, and they allow the construction of batch processes. Command lines have been the natural interface in Unix until quite recently. Furthermore, batch processes constitute a valuable and wide-spread use in workstations, thus the lack of the possibility of using them can be seen as a loss from the point of view of the user. As will be shown, most of the command lines share a common structure that can be exploited in order to simplify the construction of a graphical interface.

There exist graphical programming alternatives to text interfaces, however each one is specially suited for a particular task and has its drawbacks when facing the construction of a simple user interface [13].

- High-level graphical libraries: they are very powerful graphic managers, with a wide concept of user interface, and unspecific purpose. However, this power and unspecificity makes the programming task more complex [14] and a long period of training is needed to be able to master these libraries. Examples of these libraries are Qt [6,15] or wrapper libraries such as KDE [11] and Gnome [4].
- Graphical user interface (GUI) builders: the spectrum of treatable interfaces is reduced as well as the programming complexity. A graphical interface is designed visually using another graphical tool and the code implementing the desired interface is automatically generated. However, it is not yet trivial to insert our functional pieces of code within the automatically produced source and the programming complexity is still high enough (see [16] for a sample code). Examples of these tools are Lxb [10], Xforms [8,17], QtArch [18], Ez [12] and Glade [5].
- Hybrid languages: the visual concepts introduced by languages like Visual Basic or Visual C++ in Windows are still at the development stage in Unix. As far as we know, only Xview [7,19] provides such a language but it implies that the source code for the program being interfaced must be accessible, which is not always the case.
- Scripting languages: these languages are often interpreted with a loose data-typing. For this reason, the development task is slightly easier. However, they suffer from the same problem as all the other alternatives: in order to provide general-purpose facilities the complexity must still be high. Examples of scripting languages are Tcl/Tk [9,20] and Python/Tkinter [21].
- Special GUI languages: there are languages such as Meta-GUI [13] which have been especially designed for rapidly prototyping GUIs with a certain characteristic. In this case, Meta-GUI is intended for managing branded GUIs, all with the same look, easily maintained and at a low programming cost. Colimate falls into this category. The underlying idea is that the knowledge of the kind of interfaces being addressed causes a reduction of the programming complexity

since a specially designed GUI language can take care of all the graphical details, leaving more time to the programmer to concentrate on other issues.

As we can see, the need for graphical interfaces, easily programmed and keeping the batch processing possibility is not fully met by any of the technical choices above. There are cases where the programming language has no graphical orientation at all, like Fortran, and in other situations adapting the source code to a graphical interface in cases when the source code is available may turn out to be extremely costly.

In this paper, we will show how the common features of standard command lines can be exploited in order to provide an environment where graphical interfaces for them can be implemented at a very low programming cost. Some extra features like menu reusability, syntax inheritance, semantic prechecking and online help can make such an environment particularly helpful and simple when describing a graphical interface for a command-line program. As a drawback, any GUI out of the structure defined by the new GUI language cannot be directly addressed, although, with the present language syntax, this case is quite rare when talking about interfaces previously based on a command line.

COMMAND-LINE SYNTAX MODEL

In this section a model for command lines is proposed based on a combination of common and repetitive structures. Most of the command lines actually used fit into this model, thus the only task left is to provide a graphical framework for it (this is done in the next section).

Basically, a command line can be thought as a program command name followed by a number of arguments whose order may be important or not. In Backus-Naur Form (BNF) notation the command line syntax can be expressed as follows:

```
<command_line> ::= <program> <arg_list> {"<" <valued_arg>}
                                        {">" <valued_arg>}
<arg_list>     ::= <arg>*
<arg>          ::= <flag> | <valued_arg>
<valued_arg>   ::= <id> | <VAR>
```

where the BNF definitions of `<program>`, `<flag>`, `<id>` and `<VAR>` are not given but `<program>` is the program name, `<flag>` corresponds to the commonly used `"-flag"` or `"\flag"`, `<id>` to the intuitive idea of a fixed identifier and `<VAR>` stands for a variable, like a filename, taking a different and suitable value each time the program runs.

The previous grammar states that a `<command_line>` is formed by a program name and a list of arguments. Eventually, the standard input or output can be redirected from or to a file. The filename for the redirection is given by a `<valued_arg>`. Arguments are divided into flags (those starting with '/' or '-'), identifiers (those fixed arguments remaining the same at every run) and variables (those arguments changing from run to run, in the implemented language these variables are identified by a starting $ character). A `<valued_arg>` is a terminal argument that is not a `<flag>`, i.e. it is either a fixed identifier or a variable.

Once a certain command line syntax is defined in the grammar shown, any command line of its kind can be issued to the system shell just by asking the user for the appropriate values of the variable arguments. However, several considerations must be taken at this point.

- Not all possible parameters in a command line are needed at a time. For instance, `"tar -c $TARRED_FILE $SOURCE"` is a valid construction for Unix `tar` but so is `"tar -v -c $TARRED_FILE $SOURCE"`, and the choice of one or the other depends on the user selection. This choice option is introduced into the model by allowing `<OPTIONAL_ARG>`s to be considered (see the model below).
- There are mutually excluding arguments: if one appears the other cannot and *vice versa*, for example, flags `"-c"` and `"-x"` in Unix `tar` as they stand for creating and extracting respectively. This can be modelled by `<EXCLUSION_ARG>`s.
- There are arguments whose values are restricted to a limited set: no other values are allowed. For instance, in Unix `dd` a valid command line is `"dd conv=$KEYWORDS"` where `$KEYWORDS` can take any of the following values: `ascii`, `ebcdic`, `ibm`, `block`, `unblock`, `lcase`, `notrunc`, `ucase`, `swab`, `noerror` and `sync`. This can be achieved by the use of `<LIST_ARG>`s.
- There are mutually including arguments, i.e. those they always appear together. As an example, a certain program may normalize images following different methods. Some of them require specific parameters that appear only in the case when the associated normalizing method is chosen, and always when it is chosen the required specific parameters must be provided. This is achieved by the inclusion of both arguments in the same `<OPTIONAL_ARG>`.
- There are arguments whose presence implies the presence of another: if A is present then B is present, but it is not always true that B is present if A is. The following example illustrates the idea although it is not strictly true. The Unix linker `ld` can be forced to link with a certain library (`-l`) at a specified directory (`-L`). The directory is not needed for all libraries as there are some directories by default; however, if a directory is given some special library must come from it, otherwise the directory specification is useless. This can be done by the inclusion of an `<OPTIONAL_ARG>` inside another `<OPTIONAL_ARG>`.
- Arguments can gather into sets describing a single object: for instance, if a set of programs deals with raw images, for every single image argument they all need to know the filename and the X and Y dimensions. This is modelled by the introduction of `<USER_TYPE_ARG>`s that are single variables representing a list of arguments. User types can inherit from other user types, thus bigger argument aggregations can be built in a very simple way. Following the example with images, a certain program always receives an input image with its type as in '`-i$IMAGE_FILENAME -type $IMAGE_TYPE`', where `$IMAGE_TYPE` can be raw, gif or jpg. However, if it is a raw image then the X and Y dimensions must also be provided.

After all these considerations, the command-line model becomes

```
<command_line> ::= <program> <arg_list> {"<" <valued_arg>}
                                        {">" <valued_arg>}
<arg_list>     ::= <arg>*
<arg>          ::= <flag> | <valued_arg> | <OPTIONAL_ARG> |
                   <EXCLUSION_ARG> | <LIST_ARG> | <USER_TYPE_ARG>
```

```
<valued_arg>    ::= <id> | <VAR>
<OPTIONAL_ARG> ::= <arg_list>
<EXCLUSION_ARG>::= <arg_list> <arg_list>+
<LIST_ARG>      ::= <valued_arg>+
<USER_TYPE_ARG>::= <VAR>
```

The changes with respect to the grammar, shown previously, are introduced to provide argument flexibility. Terminal arguments are still divided as flags and valued arguments (either fixed identifiers or variables). However, meta-arguments are created to account for the optional arguments, arguments with a fixed set of possible values, mutually excluding and mutually including arguments, user-defined argument groups, etc. The flexibility achieved by this extension of the grammar covers most of the commonly used command lines.

PACKAGE AND GRAPHICAL MODEL

The command-line syntax model described above cannot form by itself a base for a GUI. Instead, it is integrated into a higher-order model defining the relationships between different programs, menus and their integration into a whole package. The objects in this more global model are: package, group, program, command line, menu and argument. All programs considered form the package for which the GUI is designed, however programs may be grouped in the package according to their different functionalities. For instance, two possible groups for Unix programs might be 'Disk information' and 'File information'; in the first group programs like df or quota can be included, while in the second we might have ls and file. However, a single program might belong to several groups since its functionality may be seen from different points of view (this would be the case of program du in the previous example). Every program shows a single menu where its arguments are prompted (a menu can be thought of as a list of arguments to be prompted). However, menus are separate entities, thus several programs may share a menu; the exact mechanism for doing this is explained below.

A single program may have several intrinsically different functions, needing for each one a substantially different set of arguments. This is the case of the Unix C compiler (cc): the arguments for compilation include directories, external definitions, the source code, the warning and error levels, code generation options, etc., while linking parameters define the external libraries, where to find them, object combination options, etc. These different functions are modelled by the fact that a program can have different command lines, each one containing all possible arguments used for that function. Several arguments may be shared among various command lines, even from different programs. In fact, this is the way menus are shared by programs. The arguments are considered to belong to the menu and not to the program. In this way any program or command line is built from arguments that are shown in a specific menu and so they can be shared among command lines within the same program or among programs.

The graphical interface must let the user navigate among the different groups, programs and command lines to select the desired function. Once a function (command line within a program) is chosen the appropriate menu must be shown, the required fields filled and then the selected command line constructed and launched in the system. This is why the original program need not be reprogrammed to introduce the graphical interface and why the batch processing feature is not lost within this interface model. In fact, the graphical interface can be a help to build batches with long and complex command lines.

EXTRA FEATURES

Aside from the core model presented above, further desirable requirements have been considered. For instance, online help on the meaning of arguments, the program functionality and use, and even some way of accessing the program Web page.

Some data typing is also included so that variables are prechecked before launching the command. For instance, if a certain argument must be an existing file, the graphical interface checks that it really exists before the command line is launched to the system. Other checks about numerical validity, string length, letter case, etc. are taken into account.

Moreover, default values for arguments may be required, this not only includes fixed default values but default values based on the content of other variables. This is achieved by the inclusion of an expression evaluation mechanism for the default value. In this way, we can set variable $A to the value $B + 1, the concatenation of $B and $C, the length of variable $B, the extension of the filename stored in variable $B or any thinkable expression combination.

Furthermore, there are variables that seldom change among sessions; the current status of all variables is stored so that they keep their values for the next time the interface is used, thus avoiding the necessity of being retyped.

COLIMATE

Colimate (the COmmand LIne MATE), is a language where the package, command line and graphical model presented so far has been built. All model features find a direct translation into Colimate. The result is a powerful language for command-line-oriented graphical interfaces where complex menu structures can be defined with very few source code lines. It must be borne in mind that this language has been designed to simplify the description of a certain kind of graphical interfaces; thus its action is strictly limited to this model, although the model is wide enough to hold a wide range of command-line interfaces.

The interface description is first compiled into an internal representation of the graphical model, and at this moment a colimate syntax and interface description consistency check is performed. Objects can be referred to before they are declared, which simplifies even more the interface description.

Once the interface is compiled, it can be run with the help of an interface generator that raises the needed windows, attends to user selections and finally launches the desired processes. The graphical interface has been developed with the Qt library, whose Unix edition is free and is part of the basic configuration of a Linux machine. In spite of the fact that the language has been conceived for Unix platforms, it has been fully ported to Windows.

Colimate also provides a graphical mechanism for a manual selection of files, and it is prepared to interact with other user-defined programs such as word editors, pdf and postscript readers, image viewers, etc.

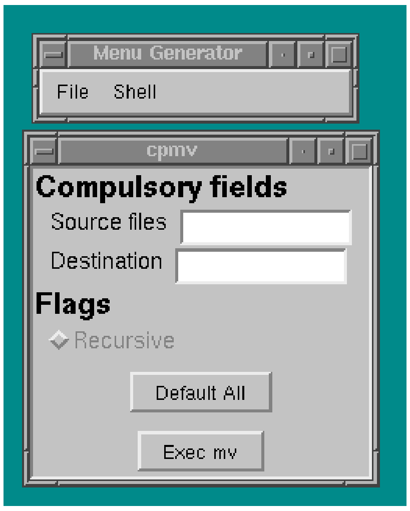

Figure 1. A screenshot for the simple copy and move interface. This window acts as a launcher for the appropriate copy and move commands: the source and destination files are selected by the user, and the command needed for that operation is run in a subshell when the ‘Exec’ button is pressed.

As a first example let us build a simple interface to the copy and move Unix commands. Only the recursive flag is considered for copy and both commands are to share the menu. The following code is the whole source code needed for the generation of the interface in Figure 1, notice its simplicity and intuitiveness. Colimate code is designed to be embedded into C and C++ programs, which is why it is written within special comments starting with `/*Colimate`. However, this embedding possibility is not a drawback for other languages since a separate file with the needed Colimate source code can always be attached to the program being interfaced.

```
/*Colimate:
   // --------------------------------------------------------------
   // Define the group ''Shell'' with only two programs
   GROUP Shell {
      PROGRAM cp;
      PROGRAM mv;
   }

   // --------------------------------------------------------------
   // Define the ''Copy'' program
   PROGRAM cp {
      // This program opens the menu ''cpmv'' that is shared with ''Move''
      OPEN MENU cpmv;
      // Define the considered command line syntaxes for ''Copy''
      COMMAND LINES {
```

```
      // Only a simple syntax with the recursive option is defined
      + simple: cp [-r] $SOURCE $DEST
   }
   // Define the parameters involved with this program’s command line
   PARAMETER DEFINITIONS {
      // The optional ’’-r’’ is an argument as any other else.
      // Its name is ’’OPT(-r)’’
      // As it is an option, no data type is required
      OPT(-r) {
         // Define the label attached to this option
         label="Recursive";
         // Define the online help for this option.
         // It appears as a yellow tag when the cursor goes by the label
         help="Allows to copy subdirectories";
      }

      // This is a variable whose type is a ’’File pattern’’
      $SOURCE {
         label="Source files";
         help="Pattern or single file";
         type=file pattern;
      }

      // The destination can be either a file or a directory although
      // the Colimate data type for both is simply ’’file’’
      $DEST {
         label="Destination";
         help="File or directory";
         type=file;
      }
   }
}

// -------------------------------------------------------------
// Define the ’’Move’’ program
PROGRAM mv {
   // It opens the same menu as the ’’Copy’’ program
   OPEN MENU cpmv;
   COMMAND LINES {
      // This is the command line syntax considered
      + simple: mv $SOURCE $DEST
   }
   // Since variables live on the menu and they have already been
   // defined there is no need to redefine them at this point
}

// -------------------------------------------------------------
// Define the ’’Copy’’ and ’’Move’’ menu
MENU cpmv {
   // This is a label separating compulsory from optional fields
```

```
      "Compulsory fields"
      // List of variables in this menu
      $SOURCE
      $DEST
      // Another label for the optional fields
      "Flags"
      // List of optional variables
      // The name of the optional variable for -r is OPT(-r)
      OPT(-r)
   }

   // -------------------------------------------------------------
   // This section specifies available actions with files. For
   // every field of the datatype ''File'' the possible actions are
   // + Manual selection of the file through a graphical interface
   // + Open an image viewer with the corresponding file
   // + Open a PDF viewer
   RIGHTBUTTON {
      + xv       : xv       $FILE
      + acroread : acroread $FILE
   }
*/
```

In this brief example the simplicity of Colimate's syntax can be seen; the programmer only has to concentrate on declaring the command-line syntax. As for the graphical interface, nearly no parameters are needed except for which variables go to which menu.

Colimate can handle very complex command lines. The features introduced in the command-line syntax model are easily expressed in Colimate.

1. Optional arguments are considered as variables whose content is a piece of command line. The name of the variable is referred to with the function `OPT()` whose argument is any of the flags or variables in the command-line piece. Furthermore, nesting of optional arguments is allowed by letting the command-line piece contain another variable representing a second optional argument.
   The following code explains how to nest options and declare them in the different Colimate sections. Figure 2 shows a snapshot of the corresponding GUI.

```
/*Colimate:
   GROUP example_group {
      PROGRAM example_program;
   }

   PROGRAM example_program {
      OPEN MENU example_menu;
      COMMAND LINES {
         // Notice the nested options
         + nested_options: prog [-i $INPUT_FILE [-r]]
      }
```

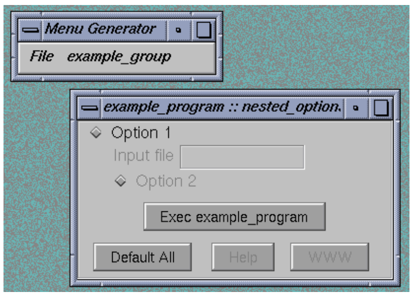

Figure 2. A screenshot for the example program with nested options. Notice that the second option cannot be selected if the first one is not.

```
      PARAMETER DEFINITIONS {
         // Notice that the three arguments must be specified in this section
         OPT(-i)      {label="Option 1";}
         $INPUT_FILE {label="Input file"; type=file existing;}
         OPT(-r)      {label="Option 2";}
      }
   }

   MENU example_menu {
      // However, only the OPT(-i) must be given to the menu
      // since it implies the rest
      OPT(-i)
   }
*/
```

2. Mutually excluding arguments (see the command-line syntax model) are managed in a very simple way. A single variable is defined for the excluding arguments. This variable can hold only one of the possible choices at a time and, thus, the exclusion is easily achieved. The following source code and Figure 3 show how this can be done in the case of a simple command line for the UNIX `tar` program. Arguments `-c` and `-x` are mutually exclusive as one creates the `tar` file and the other extracts its content.

```
/*Colimate:
   GROUP example_group {
      PROGRAM tar;
   }

   PROGRAM tar {
      OPEN MENU tar_menu;
      COMMAND LINES {
         // Variable $COMMAND stands for the excluding arguments
```

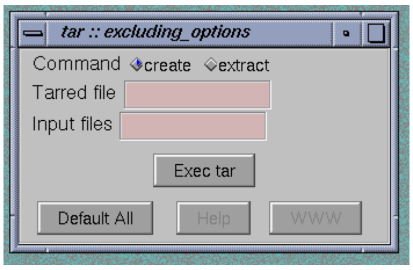

Figure 3. A screenshot for the example program with excluding arguments. Only one argument can be ticked at a time.

```
          + excluding_options: tar $COMMAND $TAR_FILE $INPUT_FILES
       }
       PARAMETER DEFINITIONS {
         $COMMAND    {
             label="Command";
             // The type of this variable is EXCLUSION. Each possibility
             // has a label, "create" and "extract", and piece of command
             // line associated. When one of the possible choices is
             // selected, then the corresponding piece of command line
             // is inserted into the launched command line
             type =EXCLUSION {
                "create"  {-c}
                "extract" {-r}
             };
          }
          $TAR_FILE    {label="Tarred file"; type=file;}
          $INPUT_FILES {label="Input files"; type=file pattern;}
       }
    }

    MENU tar_menu {
       $COMMAND
       $TAR_FILE
       $INPUT_FILES
    }
 */
```

3. User-defined types: this feature allows the user to define sets of arguments that always go together in a very simple way. Furthermore, inheritance is allowed, thus achieving very compact source codes. In the following example, an hypothetical image program takes a raw image as argument. All images are supposed to take at least two arguments, one for the image filename and another for the image filetype. Furthermore, raw images must provide the *X* and *Y* dimensions. Since all raw images are images with two extra arguments, a user-defined type `raw_image`

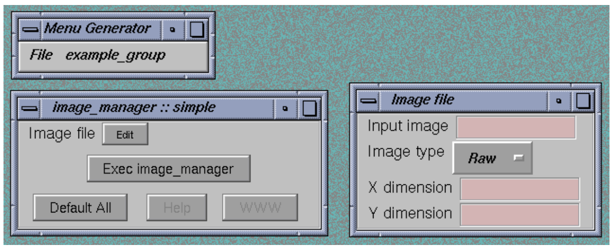

Figure 4. A screenshot for the example program with user-defined types. These complex structures are edited in a different window opened with the 'Edit' button besides the user-defined type argument.

inherits, in this case, from another user-defined type `image`. The following source code shows an example of these complex command-lines (represented in Figure 4). At the same time the use of arguments whose values are within a list is also explained.

```
/*Colimate:
   GROUP example_group {
      PROGRAM image_manager;
   }

   // Definition of the image type
   TYPE image {
      // Associated piece of command line
      // Notice the use of two IMAGE_TYPE variables: one a text
      // with the image type, and an associated list restricting
      // the type possibilities. The list variable does not produce
      // command line text, in this way it is ''invisible'' from
      // the point of view of the command line
      +: -i $IMAGE_FILENAME -type $IMAGE_TYPE $IMAGE_TYPE_LIST

      // Definition of all parameters involved in this type
      PARAMETER DEFINITIONS {
         $IMAGE_FILENAME {label="Input image"; type=file existing;}

         // The variable with the text for the image type is not
         // shown in the GUI menu. Instead, the list variable is shown
         // with its only three possible values. In this way, the user
         // is prevented from introducing wrong parameters in this
         // argument
         $IMAGE_TYPE {shown=no; type=text;}
         $IMAGE_TYPE_LIST {
            label="Image type";
            type=list {
               "GIF"  {$IMAGE_TYPE="gif";}
               "JPEG" {$IMAGE_TYPE="jpg";}
```

```
                "Raw"  {$IMAGE_TYPE="raw";}
             };
          }
       }
    }

    // Definition of the raw_image type inheriting from image
    // The piece of command line generated for a raw_image is
    // the one for images plus two extra arguments
    TYPE raw_image: image {
       +: -Xdim $XDIM -Ydim $YDIM
       PARAMETER DEFINITIONS {
          $XDIM {label="X dimension"; type=natural;}
          $YDIM {label="Y dimension"; type=natural;}
       }
    }

    PROGRAM image_manager {
       OPEN MENU image_manager_menu;
       COMMAND LINES {
          + simple: image_manager $RAW_IMAGE
       }
       PARAMETER DEFINITIONS {
         // The type of $RAW_TYPE is a user defined type
         $RAW_IMAGE  {label="Image file"; type=raw_image;}
       }
    }

    MENU image_manager_menu {
       $RAW_IMAGE
    }
*/
```

RESULTS

In this section the application of Colimate to the package Xmipp [2] is shown. Xmipp is an application for electron microscopy image processing. Its aim is to obtain the three-dimensional (3D) structure of macromolecular complexes studied with electron microscopy. For this goal it includes particle selection, 2D image processing, image classification and 3D reconstruction programs. Most of the algorithms needs many parameters to run and so far they are supplied via a command line. This possibility is quite useful for the construction of batches with well-defined tasks, but it is a burden for interactive processing. This is a typical case of a scientific package whose developers have focused on the scientific and algorithmic behavior of their programs, while the package users are biologists and biochemists who are not usually familiar with the computer textual interface. A screenshot of the interface developed with Colimate for Xmipp is shown in Figure 5.

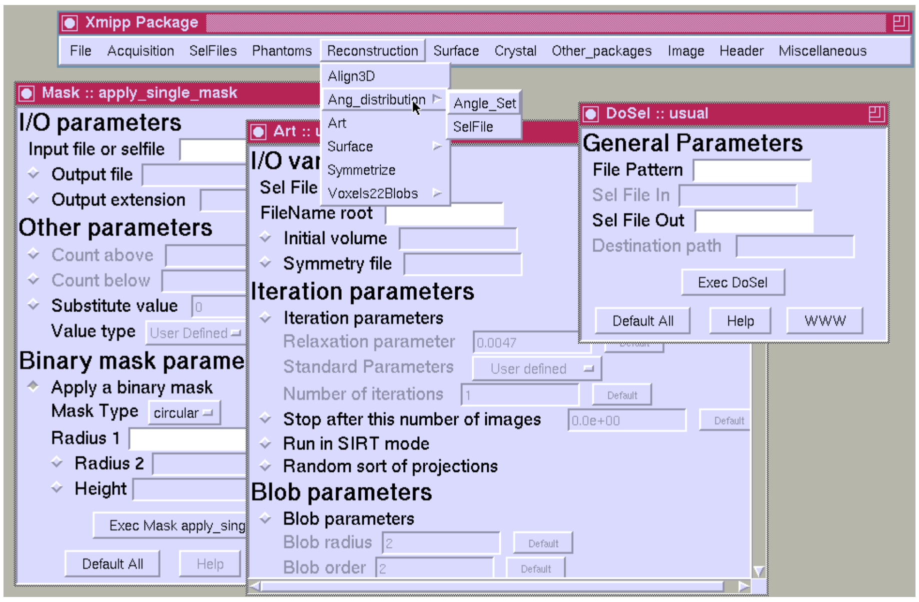

Figure 5. A screenshot of the Colimate interface for the Xmipp package.

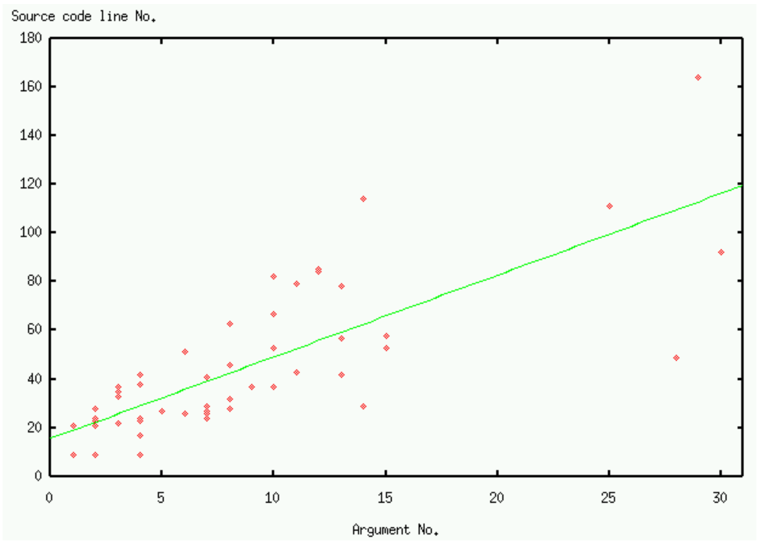

Figure 6. A plot of the number of code lines versus the number of arguments per program and its least-squares linear adjustment.

Colimate allows the construction of interfaces as complex as this one with very few lines of source code. For instance, the graphical interface for 54 Xmipp programs with 452 arguments was done with 2707 lines, which makes an average of 50.13 lines/program and nearly 6 lines/argument. A more detailed analysis of the distribution of code lines per argument is shown in Figure 6. The slope of the least-squares linear adjustment suggests that each argument can be built with as few as 3.36 lines. 50% of the programs are interfaced with less than 35 lines and 90% of them with less than 80. These figures are more than encouraging taking into account the powerful graphical interfaces provided by Colimate and the size of the source code needed if this task were to be carried out with unspecific graphic libraries and languages.

CONCLUSIONS

In this paper we have shown how the specific structure of command-line textual interfaces can be exploited in order to produce high-quality GUIs in a very efficient manner. The small size of the source code together with the easy description of the interfaces make this novel approach to graphical interfaces appealing. Further work should be done in extending the interface model to more complex structures without losing the description simplicity.

ACKNOWLEDGEMENTS

This work has been partially supported by the Spanish 'Comisión Interministerial de Ciencia Y Tecnología' through project BIO98-0761 and by the EU-FEDER programme via project 1FD97-0372.

REFERENCES

1. Frank J, Radermacher M, Penczek P, Zhu J, Li Y, Ladjadj M, Leith A. SPIDER and WEB: Processing and visualization of images in 3d electron microscopy and related fields. *Journal of Structural Biology* 1996; **116**(1):190–199.
2. Marabini R, Masegosa IM, Marco S, San Martín C, Fernández JJ, Vaquerizo C, de la Fraga LG, Carazo JM. Xmipp: An image processing package for electron microscopy. *Journal of Structural Biology* 1996; **116**:237–240.
3. Collaborative Computational Project No. 4. The CCP4 Suite: Programs for protein crystallography. *Acta Crystallographica* 1994; **D50**:760–763.
4. Gnome Consortium. Gnome user manual. http://www.labs.redhat.com/gug/users-guide/.
5. Gnome Consortium. Gnome homepage. http://glade.gnome.org/.
6. Eng E. Qt GUI Toolkit. *Linux Journal* 1996; **31**.
7. Hall M. Programming with XView. *Linux Journal* 1998; **47**.
8. Kubat K. XForms: Review and Tutorial. *Linux Journal* 1999; **22**.
9. Ousterhout J. *Tcl and the Tk Toolkit*. Addison-Wesley: Reading, MA, 1994.
10. Lxb Developers. Lxb homepage. http://www.tc.umn.edu/%7eparki005/lxb/lxb.html.
11. KDE Consortium. KDE user guide. http://www.kde.org/documentation/userguide/index.html.
12. EZ Developers. EZ homepage. http://www.ma.utexas.edu/%7emzou/EZWGL/.
13. Kotula J. Branded interface toolkits. *Software—Practice and Experience* 2001; **31**(12):1131–1142.
14. Troll Tech. Menu programming example. http://doc.trolltech.com/3.0/menu-example.html.
15. Troll Tech. Qt homepage. http://www.trolltech.com/.
16. Zhao TC, Overmars M. Xforms Designer manual. http://www.sr.unh.edu/xforms/node4.html.
17. Zhao TC, Overmars M. Xforms homepage. http://world.std.com/%7exforms/.
18. QtArch Developers. QtArch homepage. http://qtarch.sourceforge.net/.
19. Xview Developers. Xview/Openlook homepage. http://step.polymtl.ca/.
20. Tcl Developer Xchange. Tcl/Tk homepage. http://tcl.activestate.com/.
21. Tkinter Developers. Tkinter resources. http://www.python.org/topics/tkinter/.