The Astra User Interface Library

Carlos A. Furuti
furuti@dcc.unicamp.br

Projeto A_HAND
DCC—IMECC
Unicamp
Cidade Universitária Zeferino Vaz
Caixa Postal 6065
13081—Campinas SP

Abstract

Complex programs require matching user interfaces. Using basic windowing systems for implementing interfaces is a difficult task. Interface libraries, or *toolkits*, can be used to reduce programming effort while concentrating on actual application duties.

The dual text/graphics toolkit STK was proposed, emphasizing portability and simplicity. An application program should meet similar interfaces in both modules, reducing the need for separate versions designed to run in graphic and text devices. The graphical module Astra was first implemented for UNIX workstations. Some features of Astra programming and interface elements are presented. Finally, integrating libraries into a single toolkit is discussed.

Astra can be extended from the basic library. An interface editor and builder is also planned for STK programs.

Sumário

Embora aplicações complexas exijam interfaces eficientes com o usuário, sistemas gráficos básicos não são adequados à sua implementação sistemática. O uso de bibliotecas (*toolkits*) reduz o esforço de construção oferecendo elementos de interface (*widgets*) padronizados e pré-definidos.

***STK* é uma proposta de *toolkit* simples e portável suportando *workstations* gráficas e terminais convencionais. Uma aplicação usando *STK* teria um único código-fonte, podendo ser usada com interfaces similares em ambas as plataformas. O módulo gráfico Astra, já implementado e em avaliação em sistemas UNIX, é apresentado neste texto.**

Astra é flexível e pode ser estendida. Futuramente uma ferramenta poderá editar graficamente interfaces *STK* criando automaticamente código de interface para dispositivos gráficos e textuais.

1 Introduction

1.1 Writing User Interfaces

Programmers developing complex applications spend large amounts of time designing and coding user interfaces. Current graphic workstations and windowing systems allow elaborate visual metaphors closely matching actual concepts, like buttons or sliding controls.

This flexibility comes at a price—general graphic services are usually rather basic primitives so even managing a simple interface may be a long task demanding good programming skill. This effort is best shared among different applications. A *toolkit* is a collection of data and procedures implementing higher-level interface elements informally called *widgets* [9].

Some of the main benefits of using toolkits are:

- faster design as many low-level aspects are decoupled (even hidden) from the applications programmer
- simpler lay-out rearrangement (it is even possible using interface building tools like DevGuide [14] or XVT [17] for automatic code generation)
- easier learning for users due to consistent interface looks across applications; toolkits often derive from a standard interface definition (as Motif [10] or XView [7] from OPEN LOOK [13])
- resource sharing among applications, as some graphic data like colors and character fonts can be used by many programs simultaneously

And some disadvantages should be kept in mind:

- applications may become dependent on toolkit supplier for support
- graphic output other than interface rendering still requires knowledge of graphical primitives
- not all application requirements may be fulfilled by a toolkit *as is*; some extension mechanism is necessary
- on the other hand, too much generality may bring inefficiency and restrictions
- toolkits are implemented from raw graphic systems and their range of supported platforms is limited

The latter may turn to an advantage. If large programs rely entirely on a toolkit for building and handling the user interface, an otherwise difficult task of port to another computer can be made simpler by porting the (smaller) toolkit only. Given a set of lay-out specifications and an assortment of toolkit versions, an user interface generator may automatically create interface modules for the same application running in different platforms.

Although sharp cost cuts and increasing processing power have contributed to popularize graphic workstations, even lower costs and installed number keep low-end, text-based terminals important. Some customers can not afford graphic systems. And some applications, no matter how complex, can not be restricted to graphical environments[1]. So a project designed to run in different devices may be forced to split and manage versions.

1.2 Twin Toolkits

Projeto A_HAND at Computer Science Department of Unicamp is a workgroup developing an UNIX-based integrated environment for complex software development [4]. Its interests comprise distributed computing, groupware, hypertext and object-based systems, among others. As more and more projects required complex user interfaces, it was decided defining a common interface library (provisionally dubbed *STK*) for both graphic and text devices.

Ideally, the toolkit's text/graphic incarnations should have:

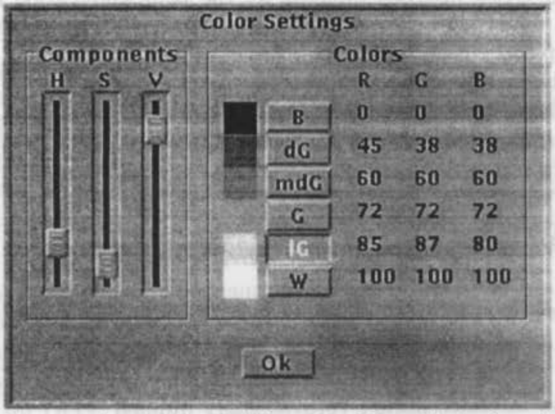
- user interfaces similar in look and function, except in essentially graphic elements
- identical application interfaces so the same program can work on different devices. This is possible using distinct libraries selected at compile time or a single universal library, set at runtime. Both approaches have strong and weak points.

STK's first version will have two separate libraries. The graphic module, called *Astra*, was the first to be implemented. Since some applications have been using it, Astra will direct and suggest its text equivalent's definition and refinement. This and other compatibility issues are addressed later.

2 The Astra Library

Astra programs are *event-driven*, as required of programs whose user interfaces offer different elements handled independently [8]. Since the user may interact in an unpredictable order, the application can not assume any usage pattern. Instead, most of time it will be prepared to answer properly to any *event*. Events are user actions, like pressing a key or moving the pointer device. Events are also asynchronous messages from the window system/manager, from other programs or from the application to itself.

[1] It is remarkable that windowing environments for both UNIX and PC systems—which contributed so much for the recent trend of new applications—are an afterthought. In contrast, other systems achieve much greater graphic integration. In Macintoshes, for instance, in all but gravest system errors an icon pops up with a dialog box prompting for user response, instead of a plain message



	R	G	B
B	0	0	0
dG	45	38	38
mdG	60	60	60
G	72	72	72
lG	85	87	80
W	100	100	100

***As-color* is a very simple Astra tool for customizing widget colors at run time in programs built using Astra. The user selects each of six basic colors used by the toolkit (by convention called from Black to White), then sets the vertical sliders changing Hue, Saturation and Value color attributes. Equivalent Red, Green and Blue percents are shown in captions to the right for display only. The color scale is a glyph. Though the selection buttons are text-labeled, they could be colored themselves, dispensing with the glyph. The tool's output is a configuration character string affecting forthcoming applications. *As-color*'s own colors change when sliders move, providing immediate feedback. Notice a 3-D visual effect looking at the page first upright, then upside-down.**

2.1 Program Structure

Environments supporting event-driven programming allow an application to create resources (like screen windows), assign which events it is interested on, then process incoming events. This last step repeatedly awaits for an event arrival and chooses an appropriate action, decided on event type, receiver object and application context.

An Astra program must perform these steps in the following manner:

1. defining *callback* or *dispatcher* functions to be executed after an event
2. requesting a connection to one or more workstations where application users may interact.
3. creating widgets and determining their looks and screen properties. Each widget category accepts a set of specific dispatchers. A programmer might set a different

callback for each possible pair widget-event, but often much fewer are needed since calling mechanisms encourage re-using functions.

A widget filters events unrelated to the application. For instance, a simulated **button** calls application code only when the user either presses or releases it using the pointer device. The application ignores actual screen drawing and other events. Usually the less specialized the widget, the more dispatchers are possible: in contrast to the **button**'s single function, a **canvas** uses four or more, since the client program may be interested in a wider range of events.

4. call a standard *event processor*. A function like *sdEventLoop ()* accepts events, finds the receiving widget's category, and

   - requests widget code to perform trivial tasks like redrawing
   - calls application dispatchers in response to specific events

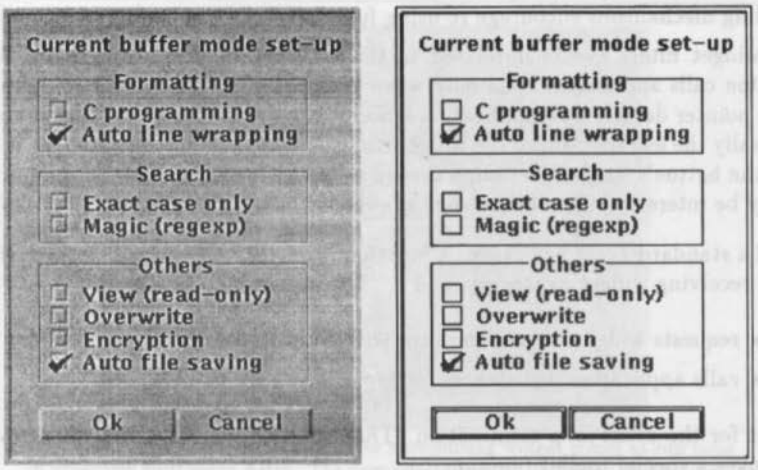
5. wait for the event loop termination. That will happen when a callback eventually invokes a special toolkit function.

2.2 Programming Levels

Astra programming is organized in levels or layers. Upper layers are easier to handle while lower ones provide stricter control over the library behavior.

- Level 0, changing the Astra library itself. Not usual, since extending Astra is possible using higher levels.
- Level 1, writing applications using widget types (all screen samples shown here are taken from programs written in C and Level 1 Astra). Widgets are data structures which the programmer partially initializes before calling a set-up routine.

  Data abstraction suffers somewhat since some access to structure members is required. However, programming is very simple and most of widgets' components need not be known. Also, Level 1 programs are well suited for automatic generation by interface editors.
- Level 2, writing applications with abstract widgets. These are now opaque structures represented and accessed by *handles*. Only library functions with polymorphic arguments set and query widget components, so Level 2 programs are more secure against library changes. It may be easily implemented as a layer upon Level 1.
- Level 3, using an interface editor. This application is an interactive tool designed to graphically assist in interface lay-out. Afterwards, a source program is automatically



Current buffer mode set-up

Formatting

- ☐ C programming
- ☑ Auto line wrapping

Search

- ☐ Exact case only
- ☐ Magic (regexp)

Others

- ☐ View (read-only)
- ☐ Overwrite
- ☐ Encryption
- ☑ Auto file saving

Current buffer mode set-up

Formatting

- ☐ C programming
- ☑ Auto line wrapping

Search

- ☐ Exact case only
- ☐ Magic (regexp)

Others

- ☐ View (read-only)
- ☐ Overwrite
- ☐ Encryption
- ☑ Auto file saving

Two snapshots of same dialog box for setting user preferences. Astra automatically adapts to color or monochrome devices; also, program and user can both set color preferences. checkboxes select "on/off" choices. Nested dialogs, like the three labeled boxes, are useful for clustering related controls.

produced with widget descriptions and callback templates, ready for integration to an application's main routines.

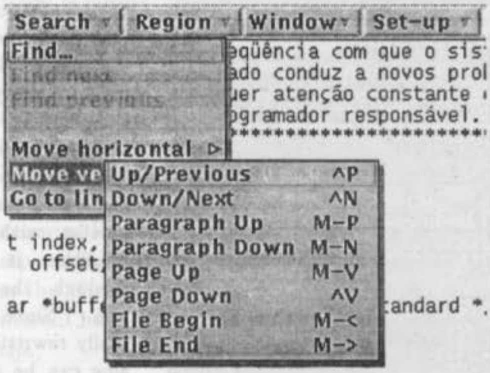
Interface editors make development easier since interfaces are frequently redesigned during program lifetime. Besides, actual coding may be postponed until an interface's efficiency and suitability are evaluated.

2.3 Available Widgets

Basic Astra widgets can be roughly organized in five groups:

dials Receiving no user input, their sole purpose is information display. Their contents are modified by the application only.

- caption: a text line
- glyph: a bitmap (two or more colors) with an arbitrary drawing
- gauge: graphically display a numerical quantity, using a linear (color strip) or circular (an arc with pointing hand) scale

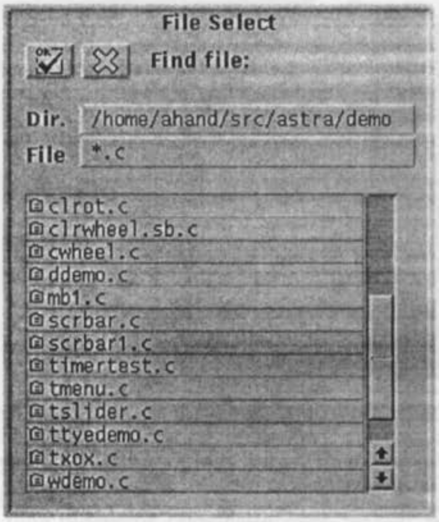


A cascade of menus hanging from a button over a fragment of ttye widget. Boxed menu items are (user-modified) defaults. Disabled entries are dimmed, and items marked with an ellipsis open dialogs. This is the first step towards a programmable editor widget. Support to accented characters is standard in text widgets.

controls Users handle these elements in order to change application data. While these are very simple, all accept at least one callback.

- button: labeled by a text string, a bitmap image or a plain colored rectangle, a button triggers its single dispatcher when the pointer device is pressed, released or both.
- checkbox: having an optional caption, checkboxes use a small "tick" mark to display an on/off state.
- slider: can control a numerical value. sliders look like a small, fixed size button which can be pushed along a linear slot. The single callback is invoked while the button is moved and/or released.
- scrollbar: albeit similar to a slider, a scrollbar has an additional *range* value. This is useful as the scrollbar frequently goes with objects with partial views like vlists and canvases. scrollbars are compound widgets since they always carry a pair of scrolling buttons. Like sliders, an application handles scrollbars using its own unit system, not raw pixels.
- textf: a line for text entry. They have scrolling and cut/paste abilities

complex controls These are more complicated widgets carrying structured data.



Dir. /home/ahand/src/astra/demo

File *.c

clrot.c
clrwheel.sb.c
cwheel.c
ddemo.c
mb1.c
scrbar.c
scrbar1.c
timertest.c
tmenu.c
tslider.c
ttyedemo.c
txox.c
wdemo.c

Astra makes simple creating compound widgets like this example, a framed **dialog** useful in many applications (it is shared by actual graphic and text editors). Comprehending **textfs**, **buttons** and a **vlist** with attached **scrollbar**, it navigates through the file system. Since titles are easily rewritten, this **dialog** can be applied for both "load" (find) and "save" operations. The small icon on the left of each list name tells directories from files.

- **ttye**: multiple lines of text. Apart some "terminal" features, like addressable cursor and emphasized text, generic text editing is left to add-ons.
- **vlist**: a scrollable vertical list. Each item holds arbitrary data labeled by a text string, an icon or both
- **canvas**: the least specialized yet most flexible widget. A **canvas** can hold and display any image and accepts most events from keyboard, pointer or both.

support frames Although nothing prevents an Astra control or dial from floating alone on the desktop screen, it is more usual for an application clustering widgets in groups with optional title and frame

- **dialog**: this widget knits together an arbitrary number of other widgets, including other **dialogs**. Moving a **dialog** moves its components; closing (hiding) or showing it hides or reveals the whole group.

  Usually a program puts its most important interface elements in a main window, visible most of time; secondary controls are set in transient windows which occasionally pop up, do their job, then hide themselves. Astra dialogs play both roles. An optional *icon* widget can be set for display while a main window itself is closed.

menus These are vertical lists designed for a quick choice. Each list item may trigger a different action (or open a child menu). A menu is able to shrink and grow. Like buttons, items can be disabled revealing an inappropriate choice for the moment. A menu can be attached under a button or pop up anywhere inside widgets like a ttye or a dialog. The same menu may be assigned to different widgets or parent menus.

Most widget categories have common attributes, like geometry and an optional help code. No on-line help is built-in; however, an Astra program may set a help callback to be invoked whenever the "Help" key is pressed or the pointer enters a widget and a flag is set. By assigning different help codes at widget creation, an application can easily provide context-sensitive help.

2.4 Special Event Handling

The five-step simple model for an Astra application, though common, is not mandatory. Instead of simply creating all necessary widgets (even if some may be invisible), calling the event loop and terminating shortly after it returns, some programs deviate in different ways.

- First of all, widgets may be created any time after a successful workstation connection. They remain intact after the event loop, which may be reentered more than once.
- An applic^tion is not limited to a single workstation. It can connect and disconnect itself to an arbitrary number of networked workstations any given time. Although a widget canno. be shared or moved across different connections, this technique does allow simultaneous application users with ease.
- While most interactive applications act only after request, some (like real-time simulations, action games, or a simulated clock) must carry on processing even in the absence of user action. So Astra permits a program to register background routines to be executed while waiting for events.
- On the other hand, the application can also explicitly call an Astra function to detect and process incoming events. This is important in lengthy callbacks, otherwise events might stay unattended till return to the event loop.

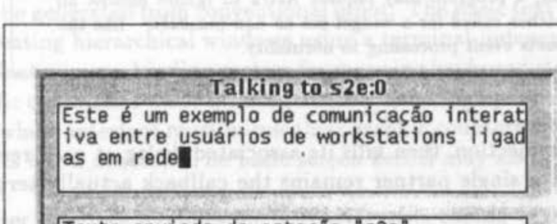
Multi-connection and background tasks are handled by the event processor in a transparent way. Low-level event management is hidden from the application programmer.

3 Level 1 Programming Example

This is a very simple UNIX *talk*-like program for on-line text exchange written in C and Astra Level 1. Most of code cares about declaring and setting widget variables. *Actual*, complex applications would dedicate a much larger share of code for callbacks and—specially—routines invoked by these functions.

```
/*--------------------------------------------------------
  Astra: an X Window toolkit - Astra Level 1 sample program

  As-talk2: a simple, two-partner only on-line text exchange

  Usage: as-talk2 remote-machine
--------------------------------------------------------*/
#include <stk/stk.h>

ttdispatcher (ttye, text, keysym)                    /* callback to execute when key is  */
Ttye *ttye;                                                 /* pressed on a "to" Ttye */
unsigned char *text;
KeySym keysym; {
  if (text)
    TtyePutChar ((Ttye *) ttye → tt_info, *text);                        /* remote echo */
}
        /* declare & partially initialize widgets (a lot is left default) */
Dialog dialog1 = {{0, 0, 310, 180}, {""}, DIALOG_FRAMED},
       dialog2 = {{0, 0, 310, 180}, {""}, DIALOG_FRAMED};
extern Ttye ttye1fr, ttye2fr;
Ttye ttye1to = {{10, 5}, {4, 40}, TTYE_FRAMED |
                         TTYE_AUTOECHO | TTYE_SHCURSOR, {ttdispatcher},
                         0, NULL, &ttye2fr},
       ttye1fr = {{10, 75}, {4, 40}, TTYE_FRAMED},
       ttye2to = {{10, 5}, {4, 40}, TTYE_FRAMED |
                         TTYE_AUTOECHO | TTYE_SHCURSOR, {ttdispatcher},
                         0, NULL, &ttye1fr},
        ttye2fr = {{10, 75}, {4, 40}, TTYE_FRAMED};
Button bt1={{0, 140}, " Hang up ", BUTTON_TEXT | BUTTON_CENTERX,
              sdButtonQuitDispatcher},
        bt2={{0, 140}, " Hang up ", BUTTON_TEXT | BUTTON_CENTERX,
              sdButtonQuitDispatcher};

main (argc, argv) int argc; char **argv;
{
  char buffer [50];
  SD *local, *remote;                                           /* screen descriptors */
  if (argc < 2 || ! (local = sdInit ("")) || ! (remote = sdInit (argv [1])))
    exit (-1);
  sdDialogSetup (local, sdRootWindow (local), &dialog1);
  sdAddTtyeDialog (&dialog1, &ttye1to, 0);                           /* actually set-up */
  sdAddTtyeDialog (&dialog1, &ttye1fr, 0);
```

```
    sdAddButtonDialog (&dialog1, &bt1, 0);
    sprintf (buffer, "Talking to %s", argv [1]);
    sdDialogChangeTitle (&dialog1, buffer);

    sdDialogSetup (remote, sdRootWindow (remote), &dialog2);
    sdAddTtyeDialog (&dialog2, &ttye2to, 0);
    sdAddTtyeDialog (&dialog2, &ttye2fr, 0);
    sdAddButtonDialog (&dialog2, &bt2, 0);
    sprintf (buffer, "Talk from %s", local → sd_name);
    sdDialogChangeTitle (&dialog2, buffer);

    sdEventSimpleLoop ();
    exit (0);
}
```



Talking to s2e:0

Este é um exemplo de comunicação interat
iva entre usuários de workstations ligad
as em rede

Texto enviado da estação "s2e"

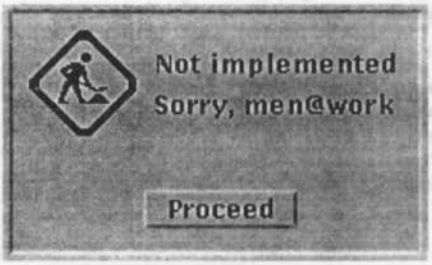
One of two dialogs created by *as-talk* invocation; the other one differs only for title's string. The upper ttye gets keyboard input echoing it to the remote partner's lower widget. Clicking on either "Hang up" button quits the application calling the predefined callback *sdButtonQuitDispatcher*.

Called with a workstation name as argument, *as-talk2* attempts to connect to both local and remote workstations. A dialog is created for each connection, containing two ttyes. Keys typed on the "to" ttye are echoed locally and (due to dispatcher action) sent to the remote partner. The "from" ttye's keyboard input is ignored.

Level 1 programs rely on widget structures, as noted in variable definitions. A Level 2 version might avoid remembering internal details, while a Level 3 editing tool would, among other benefits, make widget positioning simpler, as no screen coordinates are directly stated and aligning/spacing can be automatic.

An *as-talk* extension for connecting $N > 2$ partners was accomplished using a total of about 100 lines of code. Changes involve:

- Not two, but $N(N-1)$ dialogs are used. They are created dynamically, instead of using static variables (this is a difficult task when interface editors are used).



Not implemented

Sorry, men@work

This is a warning dialog reporting a missing feature in applications under development. It comprises captions, a glyph and a button. To prevent unwanted interaction, a program may request Astra to ignore almost all user input. A dispatcher called for a widget set as *non-maskable*—like the depicted button—resets event processing to normality.

- Button callback closes connection, then kills its associated dialog at each remaining workstation. When just a single partner remains the callback actually terminates the program.

4 Towards a Toolkit

The STK is intended to be a general and effective programming tool. Apart from event-driven style, few restrictions are posed to applications using the toolkit; also the libraries shall be no obstacle to application portability.

In short, STK is a proposal for:

- defining a simple but extensible user interface resource kit
- accelerating application development by achieving dialogue independence [6]
- enabling most client applications to run either in workstations with graphical resources or in ordinary terminals, yet presenting a similar user interface. This is its most distinctive feature and that which prompted its definition. Purely graphical toolkits, like XView, cannot be used by programs running on text-only equipment. Some user interface management/development systems, like XVT and MEWEL, are able to generate interface code for different devices (including text terminals) from a common interface definition. STK is a long-term cheaper alternative, while providing invaluable experience with interface design and development.

4.1 Work in Progress

Astra was at first the user interface module for the graphic editor Stardust [5]. After detachment from the editor, the library was much enhanced and received new widgets. Since it was implemented in C using Xlib [15] (basic library for the X Window System [12]), Astra is portable across a wide range of workstations (Stardust itself ran on Sun, IBM RS/6000, Intergraph, SCO and Interactive systems).

In the beginning of 93, Level 1 implementation was almost complete. This prompted for writing some demonstration programs, deriving widgets like a terminal emulator, and converting an Emacs-like text editor for using a graphic interface. Once ported, this will be the base for a powerful, programmable text editor widget.

Level 2 definition should start parallel to this effort.

Work now proceeds defining and implementing the text module for STK. This has different problems, since a plain terminal ordinarily lacks a pointing device and "events" must be generated from conventional input. The basic text kernel must implement services for creating hierarchical windows using a terminal-independent package like UNIX *curses* [1]. Next comes a binding system for mapping keyboard strings into abstract, application-specific commands while filtering window handling operations. Finally, the actual widgets code whose interface must mimic their graphic counterparts (except for essentially graphic objects like a canvas). The basic screen kernel may also be ported to systems other than UNIX.

User interface modules for A_HAND tools coping with hypertext [11] and groupware [3] must be reimplemented for using the toolkit.

A difficult problem to solve is setting widgets' position and dimension in a reasonable way, since plain terminals are limited to character coordinates instead of pixels. The planned interface editor should provide a solution as it is able to explore graphics flexibility while checking for text-based limitations. Exploring stretching constraints [2] might prove useful. On the other hand, the editor should yield an interface description which could also be interpreted, so that STK interfaces can be customized or reconfigured without recompiling the application.

References

[1] AT&T, *UNIX System V Release 3.2 Programmer's Guide*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

[2] Cardelli, L. *Building User Interfaces by Direct Manipulation*, Digital Systems Research Center, 1987.

[3] Castro, L. S. de, *SISTRAC: Sistema de Suporte a Trabalho Cooperativo.* Dissertação de Mestrado, DCC-IMECC Unicamp, 1991.

[4] Drummond, R. & Liesenberg, H. K. E., *Requisitos para um Ambiente de Desenvolvimento de PROGRAMAS*. I Encontro IBM de Ciência e Tecnologia da Informática, Rio de Janeiro, nov. 1987.

[5] Furuti, C. A. *Stardust—Uma Experiência em X Window*. Relatório Interno, Projeto A_HAND, DCC-IMECC Unicamp, 1991.

[6] Hartson, R. *User-Interface Management Control and Communication*, IEEE Software **6**(1), January 1989.

[7] Heller, D. *XView Programming Manual*, O'Reilly & Associates, Inc. 1990.

[8] Hartson, H. R., Hix, D. *Human-Computer Interface Development: Concepts and Systems*, ACM Computing Surveys **21**(1), March 1989.

[9] Hix, D., Hartson, H. R., *Developing User Interfaces*, John Wiley & Sons, Inc., 1993.

[10] Open Software Foundation, *OSF/Motif Style Guide Revision 1.1*, Open Software Foundation, Cambridge, MA, 1991.

[11] Polanczyk, C. A. *Uma Ferramenta Baseada em Hipertexto para Desenvolvimento de Software*. Dissertação de Mestrado, DCC-IMECC Unicamp, dezembro de 1990.

[12] Quercia, V., O'Reilly, T. *X Window System User's Guide*. O'Reilly & Associates, Inc. May 1990.

[13] Sun Microsystems, *OPEN LOOK Graphical User Interface Funcional Specification*, Sun Microsystems, Inc., 1989.

[14] Sun Microsystems, *OpenWindows Developer's Guide User's Manual*, Sun Microsystems, Inc., 1990.

[15] *XLib Programming Manual*, O'Reilly & Associates, Inc. 1990.

[17] XVT Software *XVT-Design Manual* XVT Software, Inc. 1992.