Graphic toolbox for a high level design environment

van Straaten, M.J.

Award date:
1989

EINDHOVEN UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING
DESIGN AUTOMATION SECTION

Graphic Toolbox for a High Level Design Environment

M.J. van Straaten

5 june 1989

Master thesis reporting on graduation work
by order of prof. dr. ing. J.A.G. Jess
and supervised by ir. L. Stok

Abstract

This report describes a graphics interface for a schematic editor and a simulator. Both these tools are already implemented in CommonLisp. Graphics deals with projecting pictures on a computer screen, and in the first instance one could think of a circuit editor. Ultimately, we aim at a high level design tool in which simulation of a particular module can directly be displayed by highlighting all elements in a trace set.

The base for these graphics routines is a menu handler, which opens a control window with a main synthesis menu. There are two main menus necessary: one for the schematic editor and one for the simulator. A final command selection takes place in the work window with the aid of pop-up menus, as to save space considerably. The program runs on a Apollo/Domain workstation, although it is expected that it can run on several other machines. The workstation's display may be either color or monochrome. The graphics application makes use of the CommonLisp Language X (CLX)- Interface and the X Window System as window manager.

Contents

Chapter 1

Introduction

Nowadays, the use of "Computer Aided Design"- tools (CAD) are an integral part of the design process. The Design Automation Section of the Faculty of Electrical Engineering of the Eindhoven University of Technology develops such CAD-tools, which contribute to the design of electronic circuits, particularly "Very Large Scale Integrated"-circuits (VLSI). There are several research branches in the scope of VLSI-circuits, such as the automatic- and interactive generation of VLSI-layouts, the verification and simulation of VLSI-systems, and automatic synthesis of discrete systems.

The last ten years we find an exponential growth in the complexity of VLSI-systems, which applies that "pen and paper"-methods are not feasible anymore. Workstations have become an annexe to the designer, which do not only much of the computational work but with wich designs can also directly be displayed by means of animation. These "graphic" tools have become more and more important and are often used in combination with the conventional design tools. The designer should have the possibility to redraw the symbolic circuit representation at each level in the design hierarchy. Besides, the complete contents of functional modules should have to be made visible, after which a module can be editted. The schematics entry program ESCHER (Eindhoven SCHematic Editor), developed at the Design Automation Section, fully meets this goal. The circuit's structure can be entered with the schematic editor, behaviours of modules are added using the syntax of the applicative language LISP, and the circuit can be simulated by evaluating the different modules. The simulation results are displayed using animation. ESCHER has been implemented in C, and a

new program was developed which was written in CommonLisp.

This report describes the graphics environment of this program and some of the implemented graphics routines. These routines make use of the Xwindows window system and the CLX subroutine library. All routines have been written in CommonLisp [STLE 85].

Chapter 2

Graphics application program

2.1 Introduction

When design tools make use of graphics in any way (even if they only open a window), they need a graphic interface for communication with the window system that is used. A subroutine library is commonly available which contains a set of functions that a graphic interface may use. Second, a graphics application program must include its own internal datastructure to store graphics objects which correspond with objects stored in the internal datastructure of the design tool. The following describes such a graphic interface which will be used to interface two tools: a schematics entry program and a simulator.

2.2 The graphics environment

It is proposed to provide a graphics environment which is suitable for more design tools. Therefore, it is important to develop a menu handler which can handle different programs. This is achieved by declaring some special variables at top level, which are shared among applications. These variables are bound to new values whenever another initialization module is invoked.

Once all the special variables have been initialized, the designer can start and quit an edit session at any time he wants, by mapping and unmapping the menu-handler's windows. Whenever an edit session has been quit the variables remain still be bound, so that a next edit session can easily be

started. Figure 2.1 shows the framework of such an environment.

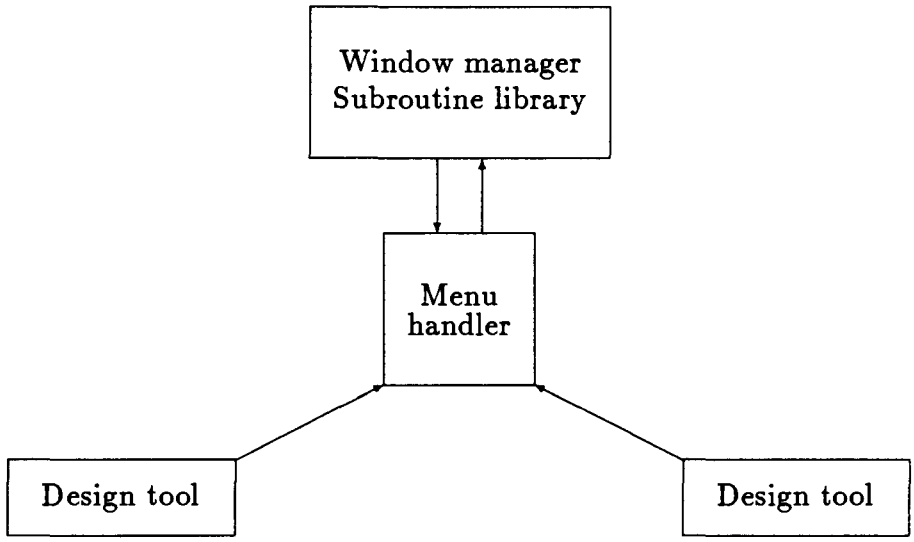


Figure 2.1: Graphics environment

2.2.1 The internal datastructure

In this section the graphic interface for one particular tool will be discussed. In most tools many design decisions have to be made, i.e. the designer makes a lot of command selections developing his circuit. Although a common menu handler is available, the graphic interface for one tool must include its own command handler that can execute the commands, which have been selected by the menu handler.

Figure 2.2 shows the datastructure for one particular tool. The common menu handler is invoked after initialization has been taken place. In this initialization module the special variables are rebound and special structures are initialized and allocated. When an edit session is started for the second time, this menu handler is directly invoked. The menu handler communicates with the particular command handler module. This command handler invokes modules of the tool, for which the graphic interface has

been written with reference to the command that has been selected in the menu handler module.

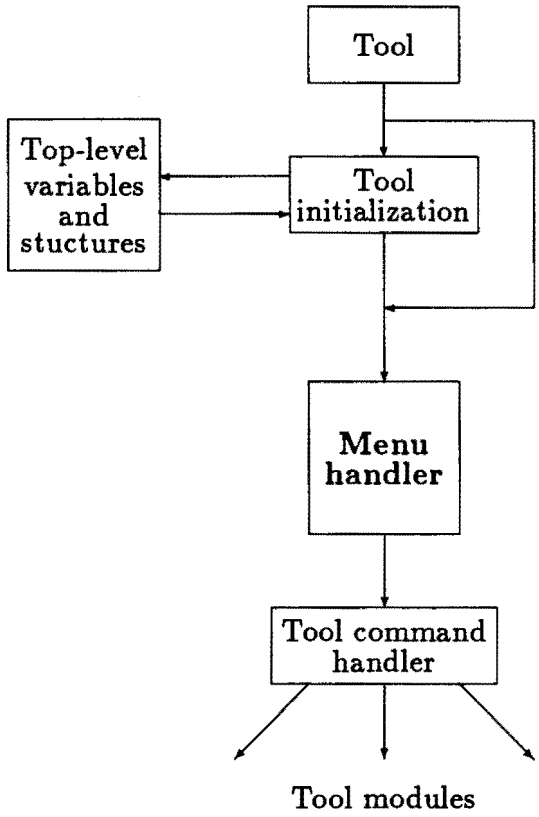


Figure 2.2: Graphics interface datastructure

2.2.2 Program loop

An edit session can be considered as one big loop. First, windows are opened and the main menu is drawn. In the second phase the user has to do a menu selection. Third, a final command selection with the aid of pop-up menus takes place and the command is executed. In the last phase is checked whether the program flag is set to true. If this is the case the loop

stops and the edit session is finished, otherwise the foregoing is repeated. In figure 2.3 the loop is shown.

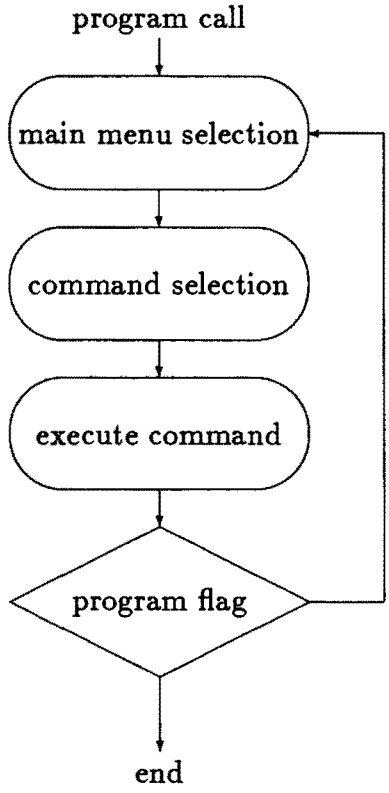


Figure 2.3: Program loop

2.3 Menu handler's global usage instructions

It is only possible to run the program in a CommonLisp environment (Domain/CommonLISP) and after the necessary files have been loaded. The

program is called using the command:

```
(hilde)
```

The following keywords may be specified: A hostname is required to open

KEYWORDS	
key	**default-value**
:hname	machine-instance
:tpage	nil

the display which by default is the machine-instance of the workstation you are currently working on. However, it is allowed to pass another hostname with the keyword :hname to open a display on another workstation. The keyword :tpage specifies whether a titlepage is to be opened or not.

The menu handler contains two main items: a main menu, which is always on the screen during an edit session, and a number of pop-up menus which appear in the work window by clicking the middle button.

The main menu is composed of small selection windows, in each of which a command text string is written. While the user enters a selection window, this window will highlighten. Leaving the window causes the window to retain its original color. So an enter-window event is visually confirmed. A main menu command is chosen by successively entering a command rectangle and clicking a mouse button. If a main menu command has been chosen by clicking the left button, a further command selection has to take place in the work window with the aid of pop-up menus.

Just like the main menu selection windows, a command selection in the pop-up-menu is visually confirmed by a color change of the filled rectangle and the text string. Whenever a further command selection has to take place, this is indicated by an arrow in the selection rectangle. The final command is chosen by releasing the middle button in a selection rectangle, not followed by an arrow. If this is not the case, a dummy command will be generated.

Although a great deal of the design can be done with the aid of pop-up menus, so that typing is avoided as much as possible, an edit window will be necessary. In this window strings can be entered in the bottomleft

corner, for instance names of templates and instances. At the same place warnings and error messages are shown, reported by the command handler after a wrong command or by the tool that is currently used. Besides, this window also shows what template is currently subject of the editor in the bottomright corner. In figure 2.4 the menu screen is represented.

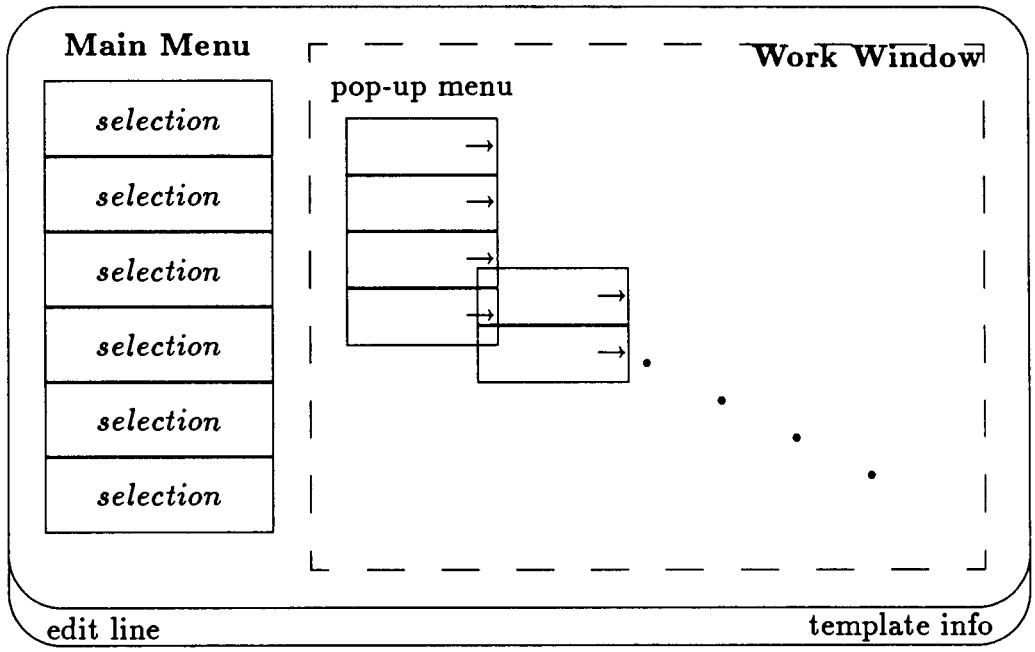


Figure 2.4: Menu screen

Chapter 3

Color Graphics

3.1 Introduction

This chapter describes the display hardware of a workstation, the use of colormaps, and attributes for color graphics. See [STEN 88] for a detailed description of displays, screens, windows, graphic contexts, graphic operations, and colormaps and colors, especially used in Xwindows. Color graphics for Apollo workstations are described in [APOL 85].

Furthermore, considerations are discussed which lead to certain graphic choices for the schematics entry program.

3.2 Display hardware

Most color displays today are based on the RGB- (red, green, and blue) color model. Each illuminated pixel on the screen is actually a mixture of the three basic colors with specified intensities. A white color appears to the human eye when the basic colors are all mixed with maximal intensity, and black appears when the three colors are turned off. Therein between a wide range of colors is possible.

In the display hardware a pixel is represented by a bitvector. The number of bits of information is equal to the number of *planes* available, which is also called the *depth*. A *pixmap* is referred to as a matrix of bitvectors: the rows correspond with the pixels in the x-direction, the columns with the pixels in the y-direction. In this way, the color display hardware can be con-

sidered as a three-dimensional bitarray or as a two-dimensional pixelarray. A one-plane pixmap is also called a *bitmap*. In figure 3.1 an eight-plane pixmap is represented. A display controller is applied to convert the digital data in the display memory to video signals that can be displayed on the monitor of your workstation.

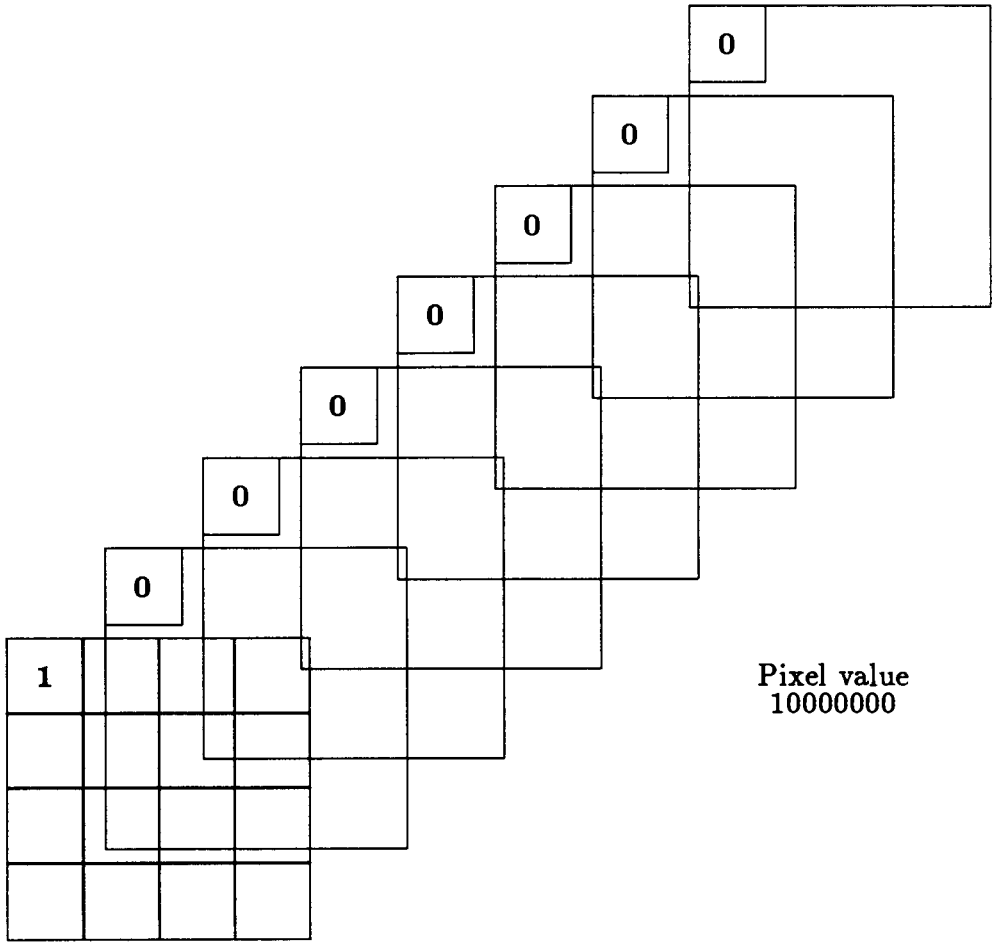


Figure 3.1: Eight plane pixmap

3.3 Colormaps

The set of rectangles with bitvectors we read in the depth forms a *raster*. In figure 3.1 the upperleft pixel in the raster has value 128. It is this value that determines the actual color of the pixel. It is clear that we need a table in which pixel values are associated with colors. Such a table is called a *colormap*, in which the pixel value numbers are used as index for color values. As we have seen earlier, a color value consists of a RGB-triple, also called a *colorcell*. The intensities of the three basic colors are usually determined by a bitvector of eight bits: 255 stands for full saturation, and 0 for no saturation. Figure 3.2 shows an example of a colormap with 256 entries, i.e. the pixmap contains eight planes. In the colormap of figure 3.2 pixel value 0 represents the color black, and pixel value 1 represents white. The colorcells belonging to the other entries, have been chosen arbitrarily. Although over 16 million different colors can be obtained in the case the intensity of the primary colors is calculated by eight bits, only 256 colors can be displayed at the same time. When an application needs more than 256 colors, more colormaps have to be constructed. Different colormaps can be installed and uninstalled any time an application wants.

In the example above we have restricted ourselves to eight bitplanes, however, there are workstations on the market nowadays which have 24 or even 32 bitplanes. It is evident that loading a colormap in the manner we did earlier is quite unfeasible for these high performance color displays (in the case of 24 planes over 16 million colors would have to be allocated!). To make the job of loading the colormap much more manageable, the available bits per pixel are broken down into three separate colormap indices: the first eight bits are used as index for red value, the second eight bits as index for green value, and the third eight bits as index for blue value.

For monochrome displays a pixel value can be represented by one bit, so a colormap has only two entries and the pixmap's depth is one. When the pixel value is 0 the pixel on the screen is black and when the pixel value is 1 the pixel is fully illuminated, or vice versa. It is often not possible to determine the brightness of the pixel by mixing the primary colors with different intensities: the pixel is either on or off.

3.4 Graphic context attributes

Most graphic packages take two items to perform graphics: One is the *drawable*, a rectangular section with a regular background pattern. This item is often referred to as a window. Second, there is the *graphic context*, in which most information about performing graphics is stored. A graphic context has a set of attributes, which specify operations that will have to be performed on the drawable. For instance, the attribute which specifies that only a certain section of the drawable will be exposed to graphic operations, is called the *clipping* attribute. Similar, we have the *line-style* attribute, which specifies in which manner lines are drawn (e.g. dashed or solid). How characters are printed is determined by the font attribute. The attribute which we will discusse in more detail, is the *raster operation* attribute. Pixel values belonging to graphic objects that have to be drawn, are calculated by these raster operations. Table 3.1 gives an overview of the possible logical functions that can be performed on foreground- and background pixels to determine the new foreground in the graphics context. Finally, there is the *plane mask* attribute, which is important with respect to raster functions. The plane mask specifies a subset of the available planes, which are exposed to any graphics operation. All other planes than specified by the plane mask are protected from modification. Plane mask is determined by taking the inclusive-or of all the planes. The default is that all the available planes are included: $2^{depth} - 1$. Formula 3.1 shows how the new foreground is calculated from the present foreground and the background:

$$\begin{aligned} new\_fgr \;=\; & ((bgr\ FUNCTION\ fgr)\ AND\ plane\_mask) \\ & OR \\ & (bgr\ AND\ (NOT\ plane\_mask)) \end{aligned} \tag{3.1}$$

The background pixels of a window or the pixels of graphic objects that have been drawn earlier are represented by *bgr*. The pixels that have to be currently drawn are represented by *fgr*, and the resulting foreground pixels by *new_fgr*. In figure 3.3 an example of a pixel value calculation is shown. In this example we assume a display with four planes and the raster function is bitwise exclusive-or. Once the pixel has been calculated as shown in figure 3.3, the value is looked up the in the colormap and the matching colorcell will drive the guns of the monitor to display the color on the screen.

Table 3.1: Logical operations on pixel values

Code	Logical function
boole-clr	All bits are assigned zero
boole-and	Perform bitwise-AND on foreground and background
boole-andc1	Perform bitwise-AND on complemented foreground and background
boole-1	Foreground is assigned
boole-andc2	Perform bitwise-AND on foreground and complemented background
boole-2	Background is assigned
boole-xor	Perform bitwise-exclusive-OR on foreground and background
boole-ior	Perform bitwise-OR on foreground and background
boole-nor	Perform bitwise-NOR on foreground and background
boole-eqv	Perform bitwise-exclusive-NOR on foreground and background
boole-c1	Complemented foreground is assigned
boole-orc1	Perform bitwise-OR on complemented foreground and background
boole-c2	Complemented background is assigned
boole-orc2	Perform bitwise-OR on foreground and complemented background
boole-nand	Perform bitwise-NAND on foreground and background
boole-set	All bits are assigned one

3.5 Graphic design considerations

Unless an application has special color needs, it should be tailored to run on any type of screen. To what extent a screen can fulfill these color needs mostly depends on the number of planes that can be used. If a screen can not provide the possibilities you have in mind, you should adjust the program that it is still applicable. First, the distinction between a color and a monochrome display should be made. Besides, decisions can be made with respect to the number of planes of a color display.

Although there are 16 different ways to calculate the foreground pixels as we have seen, we need only two functions for the schematic editor: *boole-1* (copy) and *boole-xor*. The boole-1 constant is used as function in a graphic context when objects are definitely to be placed with the specified foreground color, and the boole-xor constant is used in rubberband applications. Besides, there are only a few different colors necessary, so that only one colormap has to be constructed. This colormap is installed during

initialization and uninstalled on exit.

3.5.1 Schematic editor graphic objects

The colors we need for the main menu and the pop-up menus are rather unimportant, because a menu will never be used to draw graphic objects in it, except for the mentioned text strings. One color is needed for the background pattern and one for the text string. Obviously, we use different colors for main menu and pop-up menus, so four colors are needed for the menus.

Graphic objects in the work area, however, should be assigned priorities. For instance, a system terminal has a higher priority than the surrounding box of a module, so that a terminal should have a striking color which always lays over the less striking color of the module representation. For this reason a duplicate colorcell will have to be allocated in the colormap in the case two different objects cross each other. A new pixel value will result and object priority determines with which of the two concerning colorcells this pixel value will be associated. In table 3.2 the priorities of schematic editor objects are indicated in decreasing order. Section 3.5.2 describes in more detail the use of the available planes in association with the objects.

Table 3.2: Object priorities

Priority	Graphic object
1	Terminal
2	Gate
3	Surrounding box
4	Wire
5	Symbol

3.5.2 Plane partition

In its totality we use five colors for the schematic editor objects, four colors for the menus, black as background pattern for the windows, and white in rubberbanding. To define these eleven colors, at least four planes are necessary. When only one plane is available, obviously only black and white

can be used, but also in the case of two or three planes the display will be considered as a monochrome display, because not all the desired colors can be defined. When eight or more planes are available special techniques can be used to achieve overlays [REIL 88].

We distinguish three cases:

- 1 to 3 planes: only white and black are used
- 4 to 7 planes: there are enough planes to define all the desired colors, but there are not enough planes to use overlays for important objects.
- 8 or more planes: 5 colors can be reserved in different planes and the other planes are used to define the colors left.

Whenever a color is to be obtained, the colorname is associated with a number by means of a colortable. In the first case, negative numbers are mapped onto white and non-negative numbers onto black, otherwise the absolute value of that number is used as pixel value. For a color display, i.e. four or more planes, we define five color planes and a default plane mask. In the second case, all these color planes are bound to the default plane mask, because we can not draw in different planes. In the third case, the color plane variables are assigned different planes. Figure 3.4 shows how this is done.

The color for symbols and grid is defined in the default-plane, because symbols and grid have the lowest priority and no duplicate colorcells for this color would have to allocated. Drawing of the other schematic editor objects and the object that is currently rubberbanded, is always done in the plane which is associated with the object. The technique of overlays is represented in figure 3.5. In this example we draw a surrounding box over a terminal. However, a terminal has a higher priority than a surrounding box, so that the terminal should not be crossed by a thick line. We draw in the plane *surrounding-box-plane* as indicated in figure 3.4. The resulting pixel value is 80, and this entry in the colormap should be assigned the colorcell that resembles the terminal color. Obviously, this pixel may again coincide with a pixel belonging to an object with a lower priority, and a new pixel value will result which will also have to be allocated in the colormap as duplicate terminal color. In this way all possible combinations of pixels should have to be allocated as duplicate colorcells.

entry	red	green	blue
00000000	00000000	00000000	00000000
00000001	11111111	11111111	11111111
00000010	00000000	11111111	00000000
00000011	00000000	11110000	11110000
00000100	00000000	11110000	11111111
00000101	10000000	00000000	00000000
00000110	00000000	11111111	00001111
00000111	00001111	11110000	11111111
00001000	00000000	11111111	11111111
⋮	⋮	⋮	⋮
10000000	11111111	00001111	00000000
⋮	⋮	⋮	⋮
11111111	...	...	...

Figure 3.2: Example of a 256-entry colormap

Function: boole-xor

Plane-mask: 3

```
                  ↓↓
background        0011
foreground        1010
                  ———————— ⊕
resulting foreground  0001
```

Figure 3.3: Example of a pixel value calculation

```
*default-plane* =              7

*rubberband-plane* =           ... 10000000
*terminal-plane* =             ... 01000000
*gate-plane* =                 ... 00100000
*surrounding-box-plane* =      ... 00010000
*wire-plane* =                 ... 00001000
```

Figure 3.4: Plane assignment

Function: boole-1

Plane-mask: 16

```
                    ↓
terminal            01000000
surrounding box     00010000
                    ———————————— copy
terminal            01010000
```

Figure 3.5: Example of an overlay

Chapter 4

HILDE - High Level Design Environment

4.1 Introduction

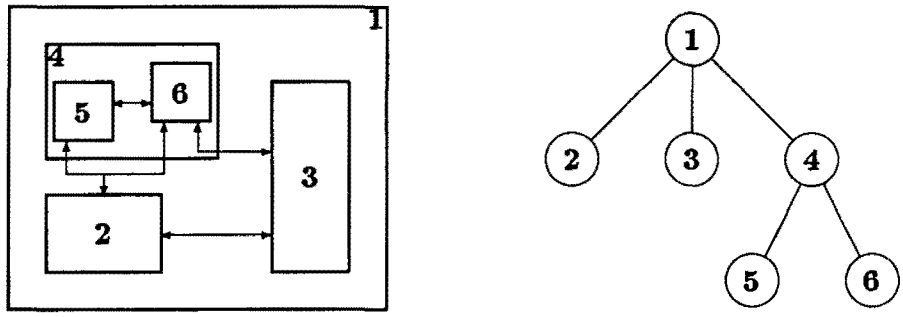
This chapter deals with the graphics datastructure for the schematic editor. Besides, the connection between the graphics datastructure and the internal datastructure will come up for discussion. A high level design environment with its datastructure was set up by [FLEU 88].

4.2 Global structure of HILDE

A way to handle complexity is to make the design hierarchical. The entire design consists of modules with interconnections and connections to the outside world. Modules in turn may consist of interconnected submodules and connections to the module itself. Figure 4.1 shows an example of a hierarchical design. Circuit 1 is composed of the circuits 2, 3, and 4. Circuit 4 is composed of the circuits 5 and 6.

The advantages of hierarchical design are obvious: the designer can fix his attention upon separate, manageable modules. Second, once the design of a module has been completed (structural or behavioural) it can be added as subcircuit in a module at a higher level in the design hierarchy. This strategy is known as *bottom-up* design. On the other hand a design environment should also support *top-down* design, so that a mixture of the

two can be used. Third, the environment should be interactive. At each step in the design a quick check could be performed whether the design step is correct.



(a) circuit representation (b) tree structure

Figure 4.1: Hierarchical design

Up till now HILDE is composed of three parts which form the basis for a high level design environment:

1. a schematics entry program to enter the circuit's structure;
2. an editor to enter component behaviours;
3. a simulator which evaluates component behaviours to simulate the entire circuit;

Graphic interfaces have to be added for all of the three programs to complete HILDE. These graphic interfaces must include a window managing mechanism, a library with graphic routines, a command handler, and a graphics internal datastructure. All these parts are described in more detail below:

- **windowing mechanism:** A design tool should have a mechanism wich creates one or more windows. These windows are mapped (made visible on the screen) and unmapped whenever necessary. For instance, the schematic editor needs a window in which the menu-handler can run and in which the schematic editor objects can be

drawn. The behaviour editor needs a window to enter a behaviour of a module.

- **command-handler:** The extension to the menu-handler which selects and executes commands of a design tool. For instance, a designer can enter and draw a complete circuit with the aid of the common menu handler and the command-handler for the schematic editor. Besides, the command-handler reports warnings and errors after a wrong command.

- **graphic routine library:** A set of functions to draw pictures, to enter strings, or to select fonts. This is an extension to the CLX-library.

- **graphics datastructure:** Although the graphics datastructure will normally be included in the internal datastructure of a design tool, these may be considered as two seperate parts: the internal datastructure contains the complete structure of a circuit, i.e. all components and the relationship between these components. The graphics datastructure only stores information about how to draw graphic objects. There is no direct relationship between the graphic objects and components in the internal datastructure to which these objects are related.

Up till now only a graphic interface for the schematic editor has been realized, so that graphic interfaces for the behavioural descriptions and the simulator remain to be developed in the future. These three graphic interfaces and the common menu-handler together form the overall graphics datastructure of HILDE. Figure 4.2 gives an overview of the global structure of HILDE. It consists of the three design tools in each of which a command-handler is incorporated. Besides, they make use of the common menu-handler to call routines in the internal datastructure and graphic routines. Vectors refer to routine calls in the direction they point to and double lines refer to data flow in both directions. Routine calls and data flow are associated with numbers in this figure and are described below.

1. Routines to create new objects in the internal structure, and to read or update existing objects. A restore routine is also included: a file is read from disc and this data is used to load the internal datastructure.

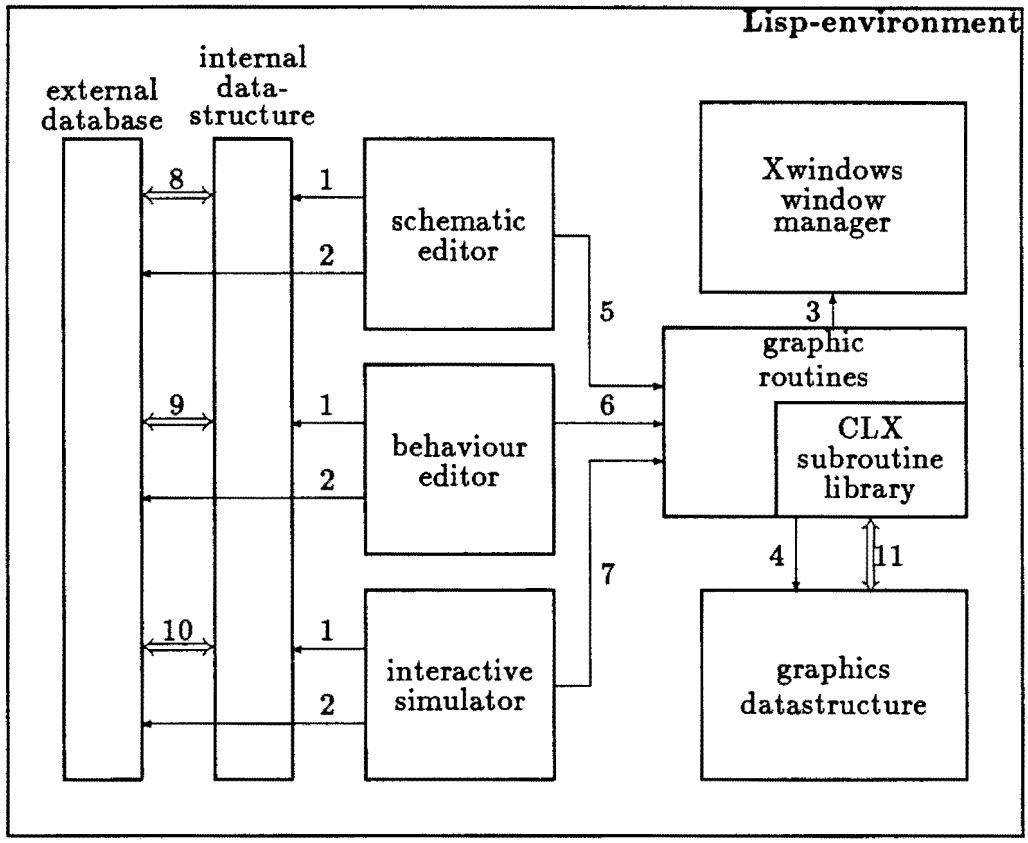


Figure 4.2: Global structure of HILDE

2. Save routine to write a file on disc. Data in the internal datastructure is converted to a file in text format.

3. Routines to map or unmap windows on the screen.

4. Graphic routines to store data in- and to restore data from the graphics datastructure. The restore routine is especially used to redraw pictures quickly.

5. The schematic editor invokes graphic routines to draw pictures in a window on the screen. These pictures are related to objects in the

internal datastructure.

6. The behaviour editor may invoke a routine to map an edit window to enter behaviours of components.
7. The interactive simulator invokes graphic routines to draw highlighted nets in a circuit or to display simulation results
8. Network data.
9. Behavioural descriptions.
10. Simulation data.
11. Graphics data.

4.3 Schematic editor

A schematic editor is a tool to define the circuit's structure. As already mentioned the circuit is entered hierarchically. A basic unit in the schematic editor datastructure is called a *template*. Templates are composed of an outer box with connections to the outside world, the *representation*, and of the *contents*. The contents of a template contains the submodules, *instances*, and the *nets* distributed among the instances. Instances contain the behaviours which are evaluated during simulation of a template.

4.3.1 Schematic editor graphics

HILDE supports bottom-up design, so that a template can be incorporated as an instance in a template at a higher level. The template's graphics consists of instances, represented by boxes with name strings in them and instance terminals, nets, and system terminals. Combinational logic is handled as follows: gates are considered as templates to which behaviours are added during initialization. When a gate is added to the circuit, the gate is incorporated as instance in the current template, and the gate's conventional picture is drawn on the screen. Furthermore, the designer has a lot of freedom to place symbols anywhere in a template: lines, rectangles, circles, arcs, and text-strings in different fonts. Figure 4.3 shows an example of

a circuit consisting of two instances. The instance "C-element" has been derived from the template C-element. When the template "C3-element" is restored (loaded from a file), the template C-element is also automatically restored. All restored templates are held in a *template-list*, so that a designer can easily change from one template to another.

4.3.2 The graphics elementtable

The elementtable is a table containing graphics information of all the objects of a particular template. Whenever the current template is changed, this table is cleared and the objects from the new template are entered. Graphics information about an object is stored in a structure. The structure *picture* is described below. An object of the type picture is created after

picture	default-value	type
id	" "	simple-string
name	" "	simple-string
funcall	nil	procedure
arglist	nil	list
position	nil	STRUCT_coord
angle1	0	angle
angle2	0	angle
radius	0	integer
font	nil	STRUCT_font

a schematic editor object has been created in the internal datastructure to which the graphic object is related, and after the object has definitely been placed in the work window, so that all information can be recorded. All graphic objects are assigned a unique identifier and a name. The slots *angle1* and *angle2* especially refer to arcs, the slot *radius* to circles, and the slot *font* to text-strings.

The information stored in the picture structure is especially useful to redraw the contents of a template: all entries in the elementtable are passed through, and with the *funcall-* and *arglist* slots a function is called and executed which draws the object that belongs to an entry.

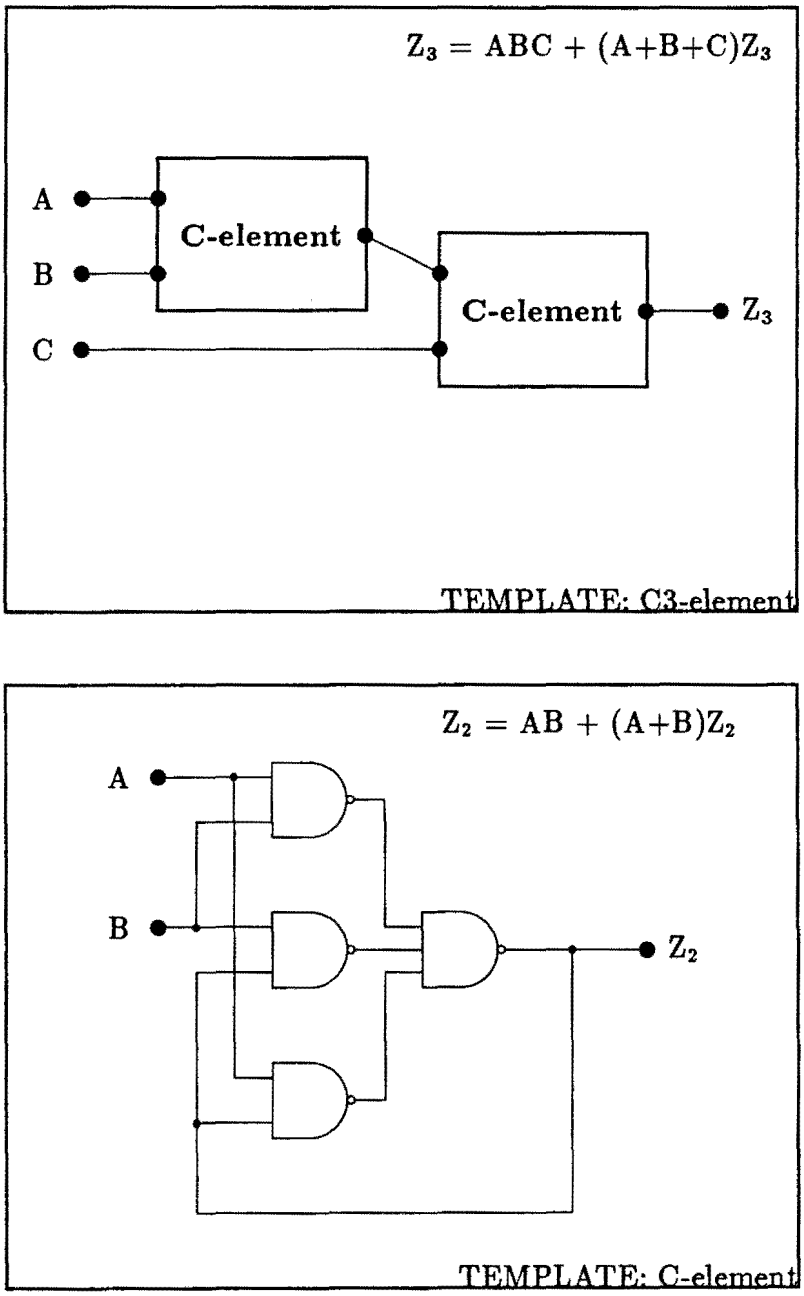


Figure 4.3: Template's graphics

Chapter 5

Xwindows Graphics Environment

5.1 Introduction

This chapter provides more global information about the X Window System, instead of discussing more detailed features at the same time. This information especially refers to the CLX CommonLisp interface of MIT [STEN 88], however, there will be a great assemblance between the C and the CommonLisp release. Most of the terms used in this report are common to all window systems. However, there may be terms which are unique to Xwindows. Therefore, it may be helpful to refer to the glossary, which is taken up in Appendix A.

Second, implementation details will be discussed using the CLX library functions. The main menu and pop-up menu structures are fully described, and the CommonLisp implementation of the *event-case*-loop in which events are dispatched.

5.2 X Window System

To achieve a graphic interface, typical library functions are necessary or at least very helpful. These functions are taken from the CLX subroutine library and are used to interface with the window system.

First of all, a display is required. A display can be considered as the

physical monitor, keyboard, mouse, and hardware of your workstation. A display is either color or monochrome (black and white) and contains one or more screens. When a screen is available, you can open an arbitrary number of windows. Besides, you can retrieve important screen information which is necessary to write your program correctly and, above all, machine independently.

The graphics application opens the computer display by calling the **open-display** function. This function requires the machine hostname as argument and returns the *display* structure. One of the slots of display is default-screen, which is normally used as single screen for the graphics interface (it is assumed that only one screen is available). The application can ask for some screen properties with the screen slots width, height, and root-depth (number of bitplanes). Besides, it can open the root window and create the default-colormap. Both these slots are structures too. In figure 5.1 a simplified display environment is shown. The root window is

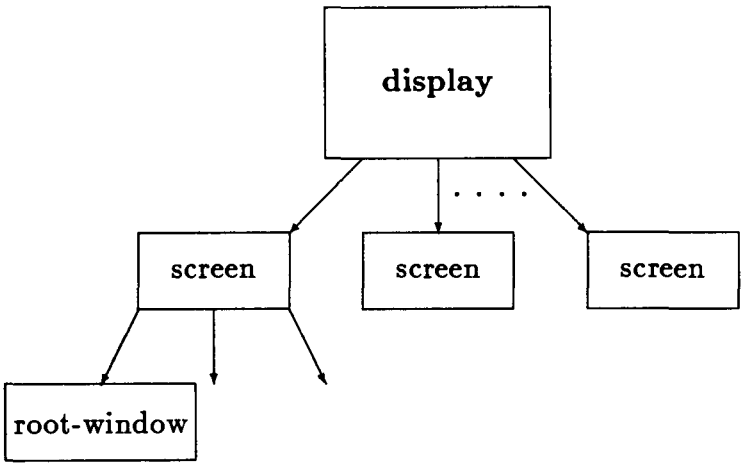


Figure 5.1: Display environment

at the top of the window hierarchy. This window covers the whole display screen, but is invisible because it can not be mapped. The direct children of the root-window are called top-level windows. Top-level windows can have children, and these children in turn may have their own children. In this way, applications can create an arbitrary deep window tree on the dis-

play screen. In the case a display provides multiple screens, such a window tree can be created on each screen. Xwindows creates a window structure when the **create-window** function is invoked, and this window can be mapped. Figure 5.2 shows the hierarchical window tree. All windows

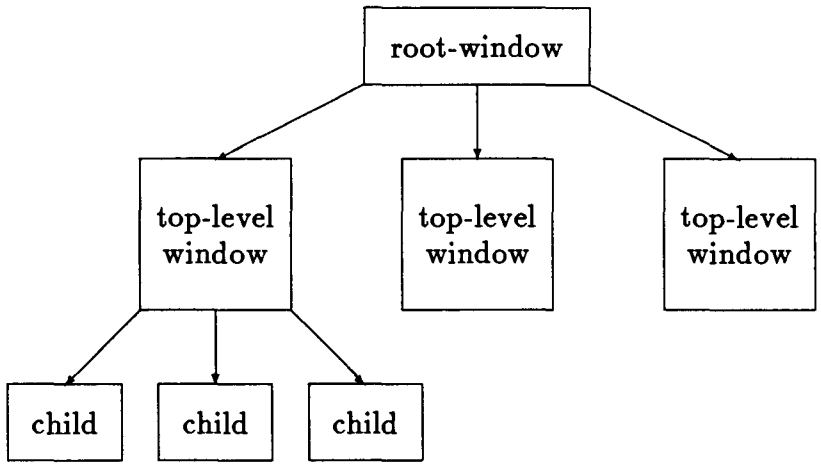


Figure 5.2: Window tree

are associated with a *gcontext*-structure which specifies particular graphics attributes, such as foreground color, linestyle, linewidth, font, and raster operation function. From here on, all kinds of drawing functions can be used to draw pictures or text.

The Xwindows default colormap is a table for wich 71 color entries are defined in the color database, and therefore 71 different colors can be obtained for a color display. Colors are stored in this table by means of a colorcell, which is a triple of three basic colors. At the same time this color is assigned a pixel value which is used as colormap entry. It is important to note that there is no direct relationship between pixel values and actual colors, so it is not desirable to use this colormap in a graphics application program in which pixel values are determined by specific logic raster operations (e.g. logical exclusive-or). Read-write colormaps, in which entries can be fixed and even be changed whenever an application wants to, are filled with colorcells by the function **store-color**. This functions needs the colormap, the colorname string or color structure, and the pixel value

as argument. The **alloc-color** function, which takes the colormap and colorname string as argument, makes an entry in the (read-only) colormap and returns the color structure with a rgb-triple that resembles the specified color, and the pixel value. This color can be used as background pattern in a particular window or as foreground in a particular graphics object or text string. For a monochrome display, the X server determines whether a specified color is mapped onto white or black.

The X server informs clients of events. Event structures are generated with reference to user input actions, e.g. pressing a key or moving the mouse. These event-types are sent completely *asynchronously* to the display event queue. A second kind of events will be generated due to mapping or resizing windows, or when a colormap is installed. These events may be considered as side effects of user requests.

In applications it is often convenient to use events as conditional tests in a loop. For instance, with reference to a button press a clause of commands must be executed. One or more event-types are often considered as being tests to quit the eventloop, for instance a button-press. So it is important to consider which event-types have to be handled in an application. An enclosure of event-types which the server will send to your application, is called an *eventmask*. Eventmasks may be created by calling the **make-event-mask** function which accepts a set of keywords as argument and returns an integer number, formed by taking the inclusive-or of all mask flags. However, the window eventmask may also be set by a list of keywords. It is important at this point that an application is prepared to handle events of all the types that are specified in the eventmask. In the case an event-type occurs in the eventmask, while it has no function in an eventloop, it must be ignored in a certain way. In this way it is prevented that the event queue will overflow with event structures.

5.3 Menu structures

Xwindows is provided with two drawable structures, *window* and *pixmap*, in which pictures or text can be drawn. The eventmask, however, is not defined for pixmaps, and therefore we must include the mouse motion event-type for this structure in the work window's eventmask. The mouse motion event has the disadvantage that a lot of events will be generated in the

eventloop when the mouse is moved. On the other hand windows have to be mapped, which is assumed to take a lot of time. An advantage of windows is that a window background can be specified and a gcontext foreground, so better looking pictures are possible. Pixmaps do not have a background, and only with the gcontext foreground one color can be specified. This foreground has to be changed, when drawing text in a filled rectangle to make the text string visible. Pop-up menus serve for further command selection, so they do not need to be visible all the time. Allocation of many pop-up menus requires much off-screen memory which, dependent on implementation, is limited. This is an important reason why we make use of a simplier structure. Furthermore, a great many commands must be handled in a schematics entry program. To reserve a selection window for each command, would require too much space and probably cover the whole screen. The use of pop-up menus, which appear and disappear on the screen to the designer's wish, saves a lot of work space. For this reason windows are used for selections in the main menu, and pop-up menus are formed by selection rectangles.

To store the underlying work area one pixmap has to be allocated for one pop-up menu, to which a part of the work-window can be copied. No pixmap has been allocated for the pop-up menu itself, so that the amount of necessary off-screen memory is kept within bounds. Pop-up menus consist of rectangles and text-strings, and are drawn in the work window at the moment the designer asks for it.

A structure is used for the main menu. The w-menu structure contains slots as menu-window, sel-window (vector of small windows), gc (graphics context), width, height, sel-width, sel-height, etc. A structure is also used for the pop-up menus. The p-menu structure contains slots such as pixmap, gc, x, y, width, height, etc. For each pop-up menu an object is created of the type p-menu. These two structures are represented in more detail in Appendix B.

5.4 User requests: Eventloops

The following describes the CommonLisp implementation of user requests with respect to reported events. Two kinds of eventloops often occur in the schematic editor and have been generalized:

- **button-press loop**: A button-press is used as conditional test to execute a certain command.
- **rubberband loop**: The shape of an object is kept up-to-date while moving the mouse, for instance lines, rectangles, and arcs.

```
(defun button-press-event-loop (function)
  (let ((delta (grid-delta *work-grid*)))
    (unwind-protect
        (do ((button-pressed-p nil))
            (button-pressed-p "Quit event processing loop")
            (event-case
             (*display* :force-output-p t)
             (button-press (event-window code x y)
                           (when (and (= (window-id event-window)
                                         (window-id *work-window*))
                                      (= code 3))
                                 (round-delta x delta)
                                 (round-delta y delta)
                                 (funcall function x y)
                                 (setq button-pressed-p t))
                           t)
             (otherwise ()
                        t))))
    ))
```

The above function in CommonLisp code is an implementation of a button-press user-request. The application asks for a button-press event and when such an event is present in the event queue (reported by the server), the body of < *function* > will be evaluated. This function will normally be a locally declared function (an unnamed lambda function or a named function declared within the *flet* special form). All object's coordinates are rounded off a grid by the macro *round-delta*. The resulting x- and y values are then passed to the body of the function. Then a boolean flag is set to t and the loop exits. The events of all the other types included in the eventmask, will be handled in the *otherwise*-clause. This clause is only an evaluation of t which is also the returned value of the clause. This

means that the event will be removed from the queue without further consequences and all other reported eventtypes than button-press are ignored in this way.

The second request we will discuss is the rubberband eventloop. Rubberbanding is used very often to size graphic objects. Whenever the mouse is moved while keeping the button pressed, the old object is removed from the screen, the shape is reculcaluted, and the object with current shape is drawn. A button-release event is used as test to exit the loop, and the object is definitely placed. The body of < *function*1 > takes care for clearing and drawing objects while moving the mouse. The body of < *function*2 > places the object and stores information in the internal graphics datastructure.

```
(defun rubber-band-event-loop (function1 function2)
  (let ((delta (grid-delta *work-grid*)))
    (unwind-protect
        (do ((button-released-p nil))
            (button-released-p "Quit event processing loop")
            (event-case
             (*display* :force-output-p t)
             (motion-notify (event-window x y)
                            (round-delta x delta)
                            (round-delta y delta)
                            (if (= (window-id event-window)
                                   (window-id *work-window*))
                              (funcall function1 x y))
                            t)
             (button-release (code)
                             (when (= code 3)
                                   (funcall function2)
                                   (setq button-released-p t))
                             t)
             (otherwise ()
                        t))))
  ))
```

5.5 Graphic performance results

This section gives a brief overview of the graphic performance of the schematic editor of HILDE using the Xwindows window system. Graphic performance can be measured at to aspects:

- Drawing speed. Depends on the efficiency of the drawing algorithms in the CLX-library and on the graphic processor that is used.
- Event processing. Depends on the connection between the X server and the client.

The speed with which graphic objects are drawn is compared to access times in the internal datastructure. For instance, when a wire is placed an object is created in the internal datastructure and routines are called to add the wire to the wire-map and to add one or two nodes to the node-map. Subsequently, a routine is called to update the netlist. Compare the add-wire command to the command that places a line as symbol. In this case only information has to be stored in the graphics datastructure. The graphics operation for both commands are the same, i.e. the draw-line routine is invoked. When no difference is observed with respect to the time to draw both lines on the screen, it can be concluded that graphics is the bottleneck. The judgement "very good" is related to the situation that the access time in the internal datastructure will be about the same as the time to perform the graphics.

The speed with which events are processed and dispatched is related to what extend the pointer on the screen can follow the mouse. When an unpercebtible delay is observed the judgement "very good" is associated with the speed of event processing.

The program has been tested on three different machines: color- and monochrome Apollo workstations and on a HP machine.

Drawing speed	
Apollo monochrome	good
Apollo color	moderate
HP color	very good

Event processing	
Apollo monochrome	moderate
Apollo color	bad
HP color	bad

Chapter 6

Conclusions

This report describes the framework of a graphics environment. New design tools can easily be incorporated in the environment, because different tools can run on a common menu-handler. This menu-handler can be used by the designer to select and execute commands of a particular tool.

Special attention is paid to color graphics of a workstation in order to achieve optimal graphic representations of objects. A graphic interface has been achieved for a schematic editor, so that circuits can be entered hierarchically. Once a circuit has been designed it can be saved on disc, and later on it can be restored and editted again.

The graphic routines make use of a window manager and a subroutine library, which contains functions to map windows and to perform graphics. The Xwindows window system which is used for the graphic interface of the schematic editor provides for a reasonable graphic performance. This performance is related to access times to the internal datastructure. However, processing and dispatching of events is intolerable slow. A great advantage of Xwindows is that portable graphics applications can be written. For instance, applications can either run on color or monochrome Apollo workstations, but applications can even be tailored to run on workstations with totally different screen properties.

All routines have been written in CommonLisp which seems to be a burden for machine and network.

Chapter 7

Recommendations

Graphic interfaces for the behaviour editor and the interactive simulator have to be incorporated in the environment as to complete HILDE.

Extend the command-handler of the schematic editor with a *select-unselect* mechanism: selected objects can be replaced, resized, or deleted.

Define a macro command mechanism: the designer can define his own commands which are equivalent to sequences of basic commands.

Bibliography

[APOL 85] *Programming with DOMAIN Graphics Primitives.*

[FLEU 88] J.W.G. Fleurkens, *HILDE- A high level design environment in CommonLisp.* Master thesis reporting on graduation work 1988, Eindhoven.

[MILN 88] W.L. Milner, *CommonLisp: a Tutorial.* Hewlett-Packard Company. Prentice Hall 1988, New Jersey.

[REIL 88] T. O'Reilly, A. Nye, *Color: A Chapter from O'Reilly and Associates Xlib Programming Manual.* O'Reilly and Associates Incorporated 1988.

[STLE 85] G.L. Steele, *CommonLisp the language (reference manual).* Digital Equipment Corporation. Digital Press 1984.

[STEN 88] D. Stenger, *CLX- CommonLisp Language Interface for the X Window System.* Version 11, Release 2. Texas Instruments Incorporated 1988, Dallas.

[STOK 88] L. Stok, R. van den Born, *EASY: Multiprocessor Architecture Optimization.* Proceedings of the International Workshop on Logic and Architecture synthesis for silicon compilers, May 1988, Grenoble.

Appendix A: Glossary

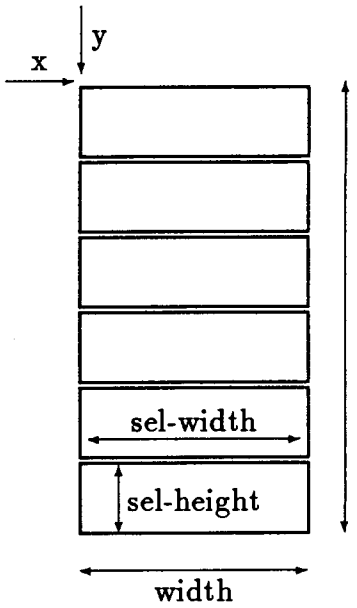
- **Ancestor**: A window somewhere nested above another window in the hierarchical window tree.
- **Backing store** : An off-screen storage of pixels of a window.
- **Bitmap**: An image that stores pixels of depth one.
- **Child**: A window that has another window as direct parent
- **Client**: A client program connects an application program to the window system server. Multiple clients make multiple paths open to the server.
- **Clipping**: This is the attribute of the graphic context, which specifies that only section of a drawable is exposed to graphic operations. Clipping areas can be specified by a set of rectangles or by a pixmap.
- **Colormap**: This is a lookup table for colors. Pixel values are associated with rgb-triples by this table. Depending on hardware limitations, one or more colormaps may be installed at one time.
- **Connection**: The path between server and client over which requests and events are sent.
- **Cursor**: Visible pointer on the screen
- **Drawable**: The destination of graphic operations. Mostly referred to as a window, however, in some cases also to as an amount of off-screen memory (pixmap).
- **Event**: Clients are informed of events by the server. Events are either asynchronously generated by input devices, or are generated as side effects of user requests. Events are grouped into event-types and are typically reported relative to a window.
- **Event-mask**: Specifies which event-types are to be sent by the server to a client.

- **Exposure:** The event-type, which informs a client that some or all of the contents of a window has been lost.
- **Font:** A font specifies how a character-set is printed.
- **Graphic context:** Information for graphics output is stored in the graphic context, such as foreground, background, linestyle, linewidth, clipping areas, and raster function. A graphic context is always associated with a drawable (window).
- **Inferior:** A window that is nested below another window in the window hierarchy. This is opposite to an ancestor.
- **Mapped:** A window is mapped when a call has been made to it, and when it is visible on the screen.
- **Monochrome:** A display is said to be monochrome, when it contains only one plane. The colomap has only two entries in this case.
- **Parent:** A direct child of a window has that window as parent.
- **Pixel:** A pixel value is a bitvector used to index a colormap to derive an actual color that can be displayed on the screen. The number of bits used to specify a pixel value is equal to the number of planes of a display.
- **Pixmap:** A three-dimensional bitarray, or a two-dimensional pixelarray. Third, a pixmap can be thought of as a stack of N bitmaps, where N represents the number of planes.
- **Plane:** A segment from a pixmap, which is two-dimensional bitarray in xy-direction.
- **Plane-mask:** Specifies a subset of planes, which can be modified by graphics operations.
- **Pointer:** The pointing device attached to the cursor, and tracked on screens
- **Request:** A single block of data, sent over from the client to the server.

- **RGB-value**: A triple which specify the intensities of the three basic colors. These values are nearly always represented as 8-bit unsigned numbers, and implemented as floating numbers between 0.0 and 1.0.
- **Root window**: The window that is at the top of the window hierarchy. This window has to be created first before another window can be mapped.
- **Screen**: The physical monitor of your workstation. Often implemented as a structure from which information can be retrieved, such as height, width, depth, and visual information. Besides, it contains the root window, from which the window tree can be created.
- **Server**: The server provides the windowing mechanism. It handles connections from clients, demultiplexes requests, and multiplexes input back to the client.
- **Sibling**: All children of the same parent.
- **Stacking order**: The order in which windows are stacked. The relationship between sibling windows is called the stacking order.
- **Visible**: A window is visible if it can actually be seen on the screen.
- **Window manager**: Manipulation of windows on the screen and policy is provided by a window manager client.
- **XYformat**: The data for a pixmap is orginized in set of bitmaps
- **Zformat**: The data for a pixmap is orginized in set of pixel values.

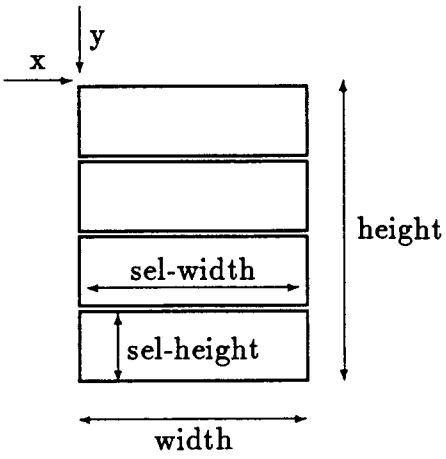
Appendix B: Menu-handler structures

w-menu	default-value	type
choice-list	nil	list
choice	0	card16
x	< *unspec* >	card16
y	< *unspec* >	card16
width	< *unspec* >	card16
height	< *unspec* >	card16
common-color	0	pixel
sel-color	0	pixel
window	nil	STRUCT_window
sel-window	nil	simple-vector
sel-gc	nil	STRUCT_gcontext
sel-width	0	card16
sel-height	0	card16
numb-of-choices	0	card16



choice-list: list containing text-strings and command-numbers for each selection window
choice: number associated with the command
common-color: pixel associated with the menu color
sel-color: pixel associated with the menu color when highlighted
window: the overall menu window; not represented for clarity
sel-window: vector containing all the selection windows
sel-gc: graphic context shared among the menu window and the selection windows
numb-of-choices: number of selections

p-menu	default-value	type
parent	nil	STRUCT_p-menu
choice-list	nil	list
choice	nil	symbol
x	0	card16
y	0	card16
width	0	card16
height	0	card16
common-color	0	pixel
sel-color	0	pixel
gc	nil	STRUCT_gcontext
sel-x	< *unspec* >	card16
sel-y	nil	simple-vector
sel-width	0	card16
sel-height	< *unspec* >	card16
inner-width	0	card16
rect-height	< *unspec* >	card16
numb-of-choices	0	card16
pixmap	nil	STRUCT_pixmap



parent: parent of current pop-up menu
choice-list: list containing text-strings and command-symbols for each selection rectangle
choice: symbol associated with the command
inner-width: max. text-string width
pixmap: off-screen storage of pixels which are copied from the work window