# The User Interface and Program Structure of a Graphical VLSI Layout Editor

*Kevin S. B. Szabó*

*Mohamed I. Elmasry*

VLSI Group
University of Waterloo
Waterloo, Ontario N2L 3G1
(519) 885-1211 x6233

**Abstract**

In this paper the user interface and program organization of the *SYMPLE* VLSI symbolic layout editor is examined. The user interface is driven by a small interpreter that is constructed from a LISP-like language at run time and has access to a consistent library of menus and graphical information-gathering functions. To improve maintainability, the editor has been constructed in a modular form with well-defined interfaces.

Keywords: user interface, CAD/CAM, vlsi editor, symbolic layout

## 1. Introduction

When designing VLSI chips, low-level *leaf* cells are still laid out manually. This is because the complexity of placing and interconnecting devices in two dimensions is currently best handled by a human designer coupled with a graphical editor. However, direct manipulation of layout geometry is not only time consuming, but error prone. Thus need arises for a design automation tool which allows the designer to simply and effectively construct VLSI leaf cells in any available chip technology. *SYMPLE*, an advanced symbolic layout editor, is such a design tool [1, 2]. Symbolic layout abstracts the details of devices and interconnection into a small set of symbols, allowing the designer to concentrate on the topology of the circuit while removing errors due to violation of the many process design rules. The symbolic notation used by *SYMPLE* is shown in Figure 1. Figures 2 and 3 show symbolic constructs that constitute devices and simple cells. The use of this notation is discussed in detail in [3, 4]. In the layout process, a designer must select, place, and interconnect a large number of symbols to create a single cell. From a user-interface designer's view, the cell editing process may be considered a task of randomly placing zero-dimensional, single-dimensional, and two-dimensional objects. These manipulations provide control over point symbols, wire symbols (for interconnection), and groups of symbols.

A block diagram of the editor is shown in Figure 4. The various modules communicate through well defined interfaces and public data types. Only the *Process Interface/Mask Generator* and *Database Interface/Compactor* share data (for reasons of efficiency). In the following sections we will examine the modules that are directly involved in the editing process; these are the user interface controller, menu, interactive input, database, editor, and graphical support modules.

## 2. The User Interface Controller

In *SYMPLE* we have taken the approach of directly configuring our user-interface through a state-machine description. Foley and Wallace [5] proposed that user-interfaces should be analyzed as state machines in order to ensure the interface possesses continuity and consistency. By providing a Finite State Machine (FSM) which describes the user-interface, the tool-developer can easily structure the man-machine communication to conform to these principles.

The FSM is built at run time from a LISP-like description (however, there is no LISP interpreter). The system requires approximately 4.5 seconds (clock on the wall time, VAX 11/785) to build the FSM for a large 65kbyte (2.5k line) configuration file. Figure 5 and 6 show the FSM description and the resulting machine. The LISP-like syntax has a number of useful characteristics; the main features of interest are its simple and easy-to-parse syntax, its extensibility, and its readability. The configuration file is compiled into an internal data structure that is efficiently interpreted. Each **state** of the machine contains either a menu, a call to an interaction routine, or a call to an application/editor routine. Menus become active when their associated state becomes the current state; menus are not shared between states. Menu contents and state transitions are described in the same file. The coupling between menu contents and state of the user interface greatly enhances maintainability of the interface, and it explicitly shows the actions associated with a menu choice.

The FSM provides the binding between menus, input functions, and data manipulation of the application. Encoding the user interface in this form has proved to be very flexible. State transitions, menu contents, and even the whole look of the system, can be modified without any source code changes. Severing the interaction and application has prevented the user interface from being scattered throughout the system. This centralization improves the consistency of the interface and the maintainability of the system as a whole. It also encourages development of interaction tools that are available to all modules of the system. A very powerful feature of the system is its ability to be reconfigured as a batch-oriented system. The input and menu modules provide some simple text-based interfaces which, when used with the FSM controller, provide a command-language (script driven) interface to the editor. This quality is important for our tool because automatic IC layout systems, such as the Icewater Silicon Compiler [6] can use our editor as a silicon assembler (providing the symbol-to-mask post-conversion). It also allows future systems to build symbolic cells by driving the editor, in a software-tool fashion, with scripts.

At present, the FSM also provides typed variables for primitive communication between states, and a state-stack. Variables can contain integers, reals, or character strings. The FSM also provides arrays of these types and the facility to declare a variable read-only (i.e. constant). The ability to call sections of the state machine as subroutines is supported with the directives **pushstate** and **popstate.** These are primarily used by the help system, which must return to the same user state as it was invoked from.

| B/W | Colour | Symbol |
|---|---|---|
| ○ | ○ green | Emitter |
| × | × red | Base |
| ▽ | ▽ green | Collector |
| ✳ | ✳ black | Schottky |
| ⊠ | ⊠ red | IIL Injector |
| □ | □ white | VIA |
| ▭ | —— orange | Buried N+ |
| —— | —— grey | First Metal |
| ━━ | —— purple | Second Metal |
| ▨ | —— red | Polysilicon |

**Figure 1: Partial Symbol Set.**

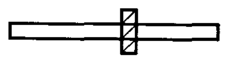
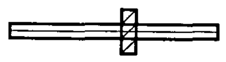
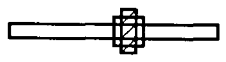
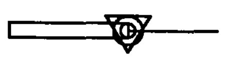
| Devices In Symbolic Form | Description |
|---|---|
| ○○○×▽ | NPN |
| ⊠ ⊠ ▽ | PNP |
| ⊠○○○▽ | IIL |
| × × ▽ | Resistor |
|  | NMOS |
|  | PMOS |
|  | N-depletion |
|  | Met, diff, Well Contact |

**Figure 2: Devices Made From Symbols.**

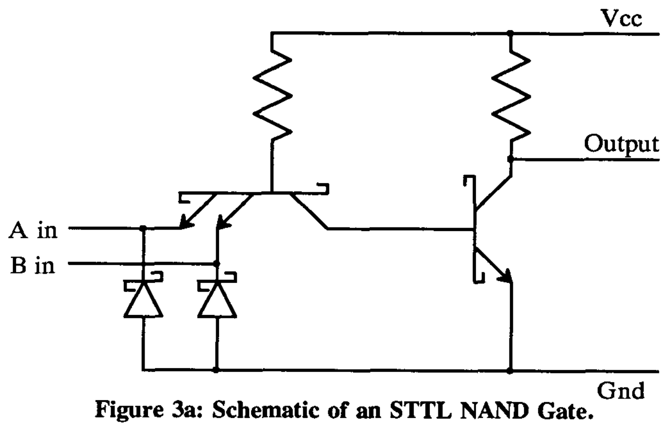


**Figure 3a: Schematic of an STTL NAND Gate.**

### 3. On-line Help

The user interface FSM provides an interactive **help** function at every user-input gathering state. The help system prototype was added in a single week, including time for the creation of help screens. When the help system was added we noticed that the system itself was a regular tree structured hierarchy with almost identical menus for each state (each menu asked what the user wanted help on, if he/she wanted more information, the next topic, etc.). This regularity allowed us to develop a help-machine compiler. The help-machine compiler takes a list of menus and their entries in a hierarchical

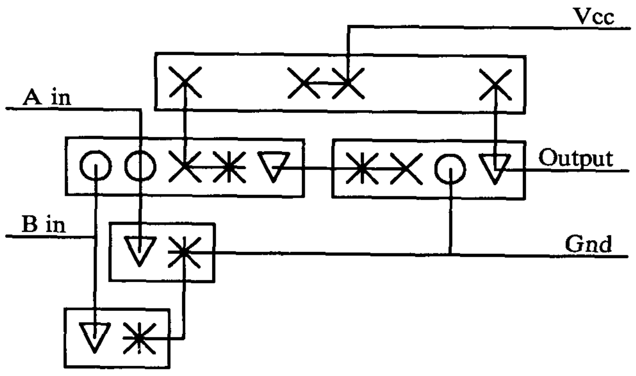


**Figure 3b: Symbolic Form of an STTL NAND Gate.**

form (figure 7); it generates a state machine with the necessary menus and states to traverse the help screens for the system (figure 9). A list of help screens required is also provided by the help-compiler (figure 8).

## 4. Menu Functions

The menu utilities interface with the user-interface controller and the window manager. They have been designed so that they may easily be replaced with system-provided menu functions that are available on most workstations. A number of menu styles are available, and new styles are easily added to the library if necessary. The basic styles available are: fixed menu, popup rectangular menu, popup round menu, popup slider (valuator) menu, and calculator-keypad menu.

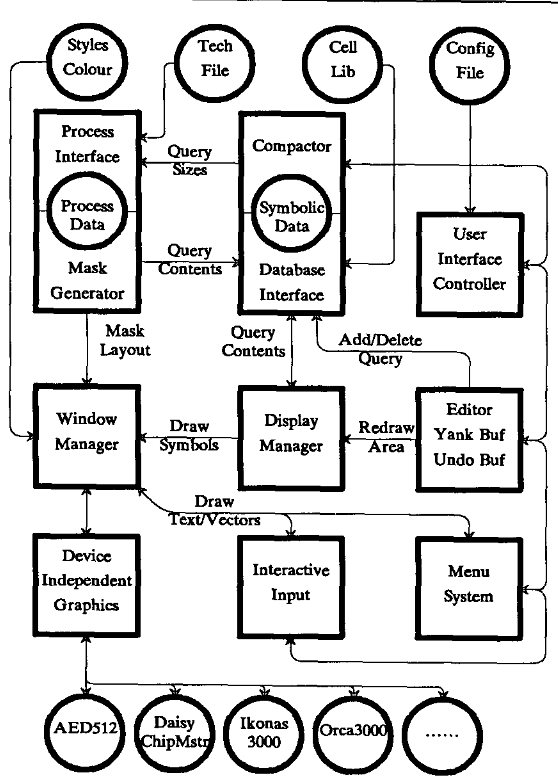


**Figure 4: Block Diagram for The SYMPLE Editor**

```
(constants (integer
        FAILURE     0
        SUCCESS     1
        LEGAL       2
        ABORT       3))
(variables
    (integer    button1) (coordinate point1))

(stateMachine
    (startState     S_BaseState )
    (stateDef S_BaseState
        (menuDef
            (menuStyle  fixed (point 0 95) (point 100 100))
            (menuHeader "SYMPLE Layout Editor")
            (menuItem (text Remove) (nextState S_rem1))
            (menuItem (text Add)    (nextState S_add1))))

    (stateDef   S_rem1
        (callFunction   PickSymbol (output point1 button1)
            (onReturn LEGAL (nextState S_remove))
            (onReturn ABORT (nextState S_homeState))))
    (stateDef   S_remove
        (callFunction   RemoveSymbol (input point1 button1)
            (onReturn SUCCESS (nextState S_homeState))
            (onReturn FAILURE (nextState S_message))))
    (stateDef   S_add1
        (callFunction   GetPoint (output point1 button1)
            (onReturn SUCCESS (nextState S_addSymbol))
            (onReturn ABORT   (nextState S_homeState))))
    (stateDef   S_addSymbol
        (callFunction   AddSymbol (input point1 button1)
            (onReturn SUCCESS (nextState S_homeState))
            (onReturn FAILURE (nextState S_message))))
    (stateDef   S_message
        (callFunction   Message (input "Operation failed")
            (onReturn SUCCESS (nextState S_homeState)))))
```

**Figure 5: User Interface FSM Configuration File**

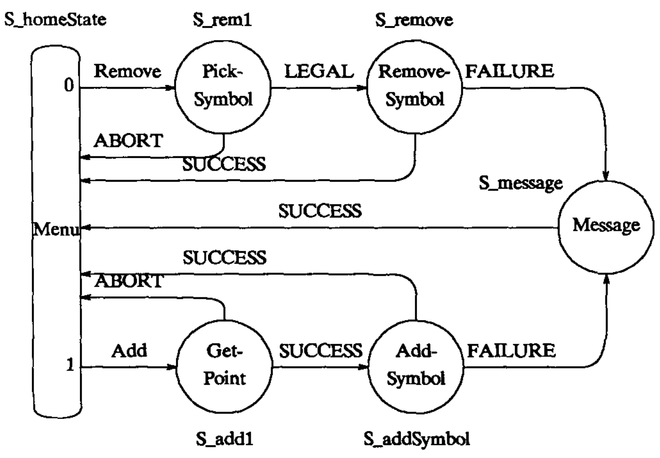


**Figure 6: User Interface FSM Described by Figure 5**

The fixed menu occupies a portion of the screen and is visible as long as its window is of higher priority than the other windows at the same location. This menu is normally used as the root of the command tree; when an item is selected its light-button remains highlighted so that any mode associated with the operation may be anticipated by the user. The fixed menus reflect state whilst the popups are fast and don't consume screen space.

```
BaseState:     remove      "Remove Symbol",
               add         "Add Symbol";

add:           success     "Successfully Get Point",
               abort       "Abort operation";
```

**Figure 7: Input For The Help-compiler**

```
/u/cad/lib/symple/help/H_BaseState
/u/cad/lib/symple/help/D_BaseState/H_remove
/u/cad/lib/symple/help/D_BaseState/H_add
/u/cad/lib/symple/help/D_BaseState/D_add/H_success
/u/cad/lib/symple/help/D_BaseState/D_add/H_abort
```

**Figure 8: List of Help Screens Required**

As mentioned previously, the design of a symbolic layout involves the unstructured placement of many symbol types; the user spends the bulk of his/her time performing this *choose and place* task. We have adapted and extended some of Baecker and Buxton's techniques [7, 8] for selecting and placing symbols. The round popup menu is used for the addition of symbols. To add a symbol, the designer points to the desired location and presses a button. All available symbols immediately pop up around the cursor without obscuring the area underneath; to select an item the user moves the cursor to the symbol and releases the button. The symbol is then added to the design. This interaction method was investigated by Singh in his Benesh Dance Editor [9]; it provides menu choices in the area of interest, avoids the large arm movements experienced in systems with a single fixed menu area, and by placing the symbols in a consistent position relative to the cursor it encourages the user to develop muscle-memory.

```
;-------------------------------------------
; Help states for "BaseState"
;-------------------------------------------

(stateDef H_BaseState
      (callFunction OutputHelpScreen
            (onReturn FAILURE
                  (nextState M_BaseState))
            (onReturn SUCCESS
                  (nextState M_BaseState))
      )
)

(stateDef M_BaseState
      (menuDef help1
            (menuHeader "What would you like to do next?")
            (menuStyle terminal (point 0 0) (point 60 15))
            (menuItem (text "Exit the Help Subsystem")
                  (nextState S_BaseState))
            (menuItem (text "Main Help Menu")
                  (nextState H_BaseState))
            (menuItem (text "More information on this topic")
                  (nextState M2_BaseState))
      )
)

(stateDef M2_BaseState
      (menuDef help2
            (menuHeader "What would you like detail on?")
            (menuStyle terminal (point 0 0) (point 60 15))
            (menuItem (text "Exit the Help Subsystem")
                  (nextState S_BaseState))
            (menuItem (text "Main Help Menu")
                  (nextState H_BaseState))
            (menuItem (text "Back to less detailed help")
                  (nextState H_BaseState))
            (menuItem (text "Remove Symbol")
                  (nextState D_BaseState/H_remove))
            (menuItem (text "Add Symbol")
                  (nextState D_BaseState/H_add))
      )
)
 -------------------------------------------

; Help states for "Remove Symbol"
;-------------------------------------------

(stateDef D_BaseState/H_remove
      (callFunction OutputHelpScreen
            (onReturn FAILURE
                  (nextState D_BaseState/M_remove))
            (onReturn SUCCESS
                  (nextState D_BaseState/M_remove))
      )
)

(stateDef D_BaseState/M_remove
      (menuDef help3
            (menuHeader "What would you like to do next?")
            (menuStyle terminal (point 0 0) (point 60 15))
            (menuItem (text "Exit the Help Subsystem")
                  (nextState S_BaseState))
            (menuItem (text "Main Help Menu")
                  (nextState H_BaseState))
            (menuItem (text "Main Menu for this group")
                  (nextState M2_BaseState))
            (menuItem (text "Next topic (Add Symbol)")
                  (nextState D_BaseState/H_add))
      )
)

; Help states for Add Symbol, Successfully Get Point,
; and Abort Operation are not shown
```

**Figure 9: The State Machine For The Help System**

The rectangular menus are sized and formated by the menu-system according to parameters that were placed in the FSM configuration file. These parameters set the style, the menu size (for popups) or position on the screen (for fixed menus), a list of text or iconic menu items, and the number of columns or rows desired. The menu is presented as an array of light buttons.

All actions provided by the menu and interaction systems can be accessed with a single button device. Use of a single button device reduces new-user confusion [10] and allows the system to work with a stylus, which is preferred by many users (e.g. artists). Common functions such as **help, abort** and **undo** can be bound to extra buttons on the input device, to menu choices in the appropriate menu, or to light buttons at a fixed point in the display. In the future we will allow keystrokes to be bound to menu choices (a single keystroke will be equivalent to a menu pick). This will provide the expert user with a two-handed high-bandwidth communication path and allow us to implement Shneiderman's B.L.T. approach [11] for fast menu selections. In the BLT approach an expert user can short-circuit the verbose menus by typing a short string of letters; each keystroke represents a menu choice. An expert user can quickly move through large hierarchical menu systems with such an interface.

## 5. Interactive Input

The interaction library contains a number of general purpose tools for gathering points, lines, and areas from the user. Lines and boxes are rubber banded and optionally snapped to a grid associated with the window. The data from these routines are communicated to application or other interaction routines by variables declared in the FSM configuration file.

The symbolic notation requires all interconnect wires to be horizontal or vertical. To ease entry of these lines, the wire interaction routine decomposes angled lines into two parts, one horizontal and the other vertical. The orientation of the lines (i.e. which piece is horizontal and which is vertical) depends on the sign of the cursor's |dx| - |dy|. In the example of Figure 10-c, the cursor has horizontal motion, resulting in rubber-banding where the wire connected to the cursor (the *alignment* wire) is also horizontal. The wire section between the jog and the start point (the *interconnect* wire) is perpendicular to the *alignment* wire. Figure 10-f shows vertical cursor motion. The net effect is that the alignment wire follows the cursor and that the wire orientation can be altered through cursor-gestures, without a menu or button choice.

## 6. The Database, Editor, and Display Manager Modules

With the exception of the compactor, all modules communicate with the database through four major routines:

```
DbAddSymbol( cell, symbol )
DbDelSymbol( cell, symbol )

DbSearchArea( cell, area, options )
DbNextSymbol( &symbol )
```

The DbSearchArea() initializes a search of the database, and successive calls to DbNextSymbol returns the items within the given area. The *symbol* type is a union of all possible symbols; it forms a basic symbol object. This interface, in conjunction with an object oriented programming approach, simplifies communication with the other parts of *SYMPLE*. For instance, a **YANK-area** operation consists of a single loop that calls DbNextSymbol and links the output onto a linked list. **UNDO** is implemented as a single **YANK-area** that occurs before the actual edit step. The **UNDO** action occurs when the yanked data is **PUT** back into the database (again, only a five-line loop). The editor itself is very easy to maintain and extend; the most complicated edit routine at this time is the **DELETE** function, shown in figure 11. To simplify the editor and allow complete flexibility in the order of symbol placement the editor does not immediately recognize illegal symbolic constructs. Symbol syntax checking and mapping of symbolic form to mask form is provided by a secondary scan of the completed layout [2].

The **Display Manager** is responsible for keeping the screen display in synchronization with the database. This module decouples the database and editor from the window manager. The edit code communicates **Add** and **Delete** requests to the database, and then requests the display manager to redraw the affected area. While this may result in more redrawing than necessary, the simplification of the rest of the system is enormous, and the screen handling software is centralised. The ability to redraw any area of the screen also allows the system to be used with window managers which do not retain obscured portions of the screen. The **Display Manager** also maintains multiple cell views. Each view may examine different areas of the cell in separate windows (neither the editor or database require this knowledge).

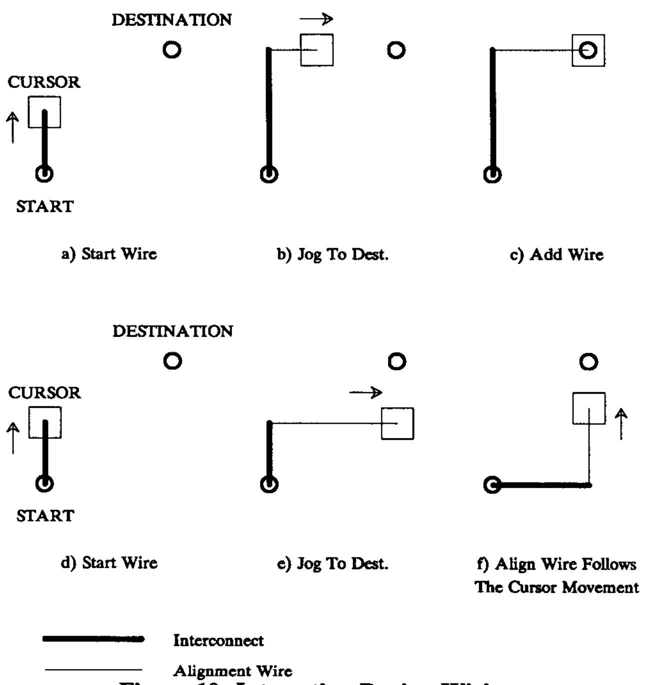


**Figure 10: Interaction During Wiring**

```
/*
 * Name.......: EdDelSymbol
 * Imports....: arg1:point1 (CM_COORD) arg2:point2 (CM_COORD)
 * Returns....: SUCCESS | FAILURE
 * Purpose....: Delete a symbol from the database.
 */
#define Window  CmGetCoord(1).window /* Parameters (variables) */
#define Point1  CmGetCoord(1).point  /* from User Interface    */
#define Point2  CmGetCoord(2).point  /* Controller             */

    int
EdDelSymbol()
{
    extern  bool             DbDelSymbol();
    extern                   Ed_yank();
    extern  ED_YANK_BUF      Ed_udBuffer;    /* Undo Buffer */
    extern  ED_YANK_BUF      Ed_ykBuffer;    /* Yank Buffer */
    ED_SYMBOL                *symPtr;
    RECTANGLE                searcharea;

    /* ensure the point is in the proper window type */
    if( cell = DiWindToCell( Window ) == DB_NO_CELL ) {
        return( FAILURE );
    }
    searcharea.ll = Point1;
    searcharea.ur = Point2;
    UtOrderRect( &searcharea );
    Ed_yank( cell, &searcharea, &Ed_ykBuffer );
    Ed_yank( cell, &searcharea, &Ed_udBuffer );

    symPtr = Ed_udBuffer.symList;
    while ( symPtr ) {
        (VOID)DbDelSymbol( cell, &(symPtr->symbol) );
        symPtr = symPtr->next;
    }
    DiRedraw( cell, &(Ed_udBuffer.area) );
    return( SUCCESS );
}
```

**Figure 11: The Delete Routine**

## 7. Window Manager

*SYMPLE* communicates with the graphics device through the **Window Manager** and a device independent graphics package. The graphics package was written at Waterloo and is loosely based on the GKS standard [12]. The **Window Manager** provides multiple overlapping windows and drawing operations for these windows. For ease of porting, the window system has an interface which is functionally compatible to the window systems provided with high performance workstations. Since our environment still includes many older pieces of graphics hardware, the windowing system also supports multiple windows on less-intelligent devices such as the AED512 and the Orca3000.

The window manager splits windows into a number of maximal horizontal strips; a drawing operation attempts to draw in each of these strips in turn by mapping the strip to a viewport and relying on the graphics package to clip superfluous output. This crude technique does not damage performance of our editor because our graphics terminals communicate to the host through serial lines and hence are I/O limited (clipping occurs in the host, not at the graphics terminal). In the window manager, the colour, stipple, fill-style, line-style, and write-mask attributes are all bound to *styles* in a manner similar to GKS; all graphical I/O is done in terms of *styles*. This interface improves the editor's portability because remote sites can easily tailor the *style* table to their graphical device's characteristics. It also simplifies the application code for drawing; a routine merely picks a style and starts to draw. The editor guarantees that symbolic features are drawn in a known order and thus the display can still take advantage of colour table and stippling techniques that simulate overlapping and transparency [13]. Interactive routines draw in preset interactive 'styles' that map into either a reserved bitplane or into a XOR drawing mode (for a colour or B/W display respectively). A **shape** drawing function is also provided. It provides the application code with limited segments; symbolic I/O only requires a handful of symbols to be drawn and these are ideally mapped into these segments. The shape-table is initialized at run time. The symbols may be tailored for device (or site) dependent considerations.

The window manager differs slightly from other implementations because of its support for fast menu operations. A special **overlay** window is provided for use by pop-up menus. Basically an efficiency extension, this window avoids splitting underlying windows into obscured/visible portions and attempts to operate in a non-destructive manner. On multiple bit-plane systems the overlay window is assigned to an unused bitplane; on systems with *bitblit*[14, 15] the obscured section is copied off-screen and then on-screen. If there is no hardware of either kind, the system gracefully degrades to the slower draw-menu/redraw-obscured-contents behaviour. During the time the overlay window is visible no other graphical I/O may occur. The overlay window is an important extension for keeping pop-up menus sufficiently fast for smooth interaction, especially when the display is a dumb raster device which communicates via a 9600 baud serial link.

## 8. Summary and Conclusions

It is difficult to build a user interface which accommodates a wide range of users; the verbose system is good for a novice but tedious for experienced users. A CAD system must be flexible enough to support the full range of designer expertise: the novice user, the casual user, and the expert user. *SYMPLE* supports them through unobtrusive menus, a user-tailorable interface, and extensive on-line help with various levels of detail and tutorial capability.

The user-interface controller and set of menu/interaction routines form a mini-UIMS (User Interface Management System) which is both more powerful and smaller in size than its hard-wired counterpart. It is a simple and effective user interface which is also being used as the basis for new interactive tools that are being built by the VLSI group.

A prototype of *SYMPLE* was released on the University of Waterloo CAD Tools tape in October 1986. Further information on the notation and construction of *SYMPLE* is available in previous publications [1, 2, 3, 16, 17].

## 9. Acknowledgements

We would like to thank Jim Leask, Mark Pulver, and Joe Morrison for their help in this work. Jim Leask is co-author of *SYMPLE* and shares credit for many of the ideas presented here. Joe Morrison wrote the help-system/help-compiler; Mark Pulver worked on the device-independent graphics package. We would also like to thank the members of the VLSI group, the Computer Graphics Laboratory, and the referees for their help.

## 10. References

[1] K. S. B. Szabo and M. I. Elmasry, Symple: A Process Independent Symbolic Layout Tool For Bipolar VLSI, *Proceedings of ICCD*, 1984, IEEE, Port Chester, New York, 474-47.

[2] K. S. B. Szabo, J. M. Leask, and M. I. Elmasry, SYMPLE: A Symbolic Layout Tool For Bipolar and MOS VLSI, *IEE Computer Aided Design* **(In review)**, 1987, IEE.

[3] K. S. B. Szabo, J. M. Leask, and M. I. Elmasry, Symbolic Layout For Bipolar and MOS VLSI, *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems* **(To Be Published)**, 1987, IEEE Circuits and Systems Society.

[4] M. I. Elmasry, *Digital Bipolar Integrated Circuits*, John Wiley and Sons, New York, USA, 1983.

[5] J. D. Foley and V. L. Wallace, The Art of Natural Graphic Man-Machine Conversation, *Proceedings of I.E.E.E.*, April, 1974, 462-470.

[6] P. A. D. Powell and M. I. Elmasry, The ICEWATER Language and Interpreter, *21st Design Automation Conference*, June, 1984, IEEE, 98-102.

[7] R. Baecker, Towards An Effective Characterization Of Graphical Interaction, *Seillac II Workshop On Man-Machine Interaction*, 1980, North-Holland Publishing Co. IFIP.

[8] W. Buxton, An Informal Study Of Selection Positioning Tasks, *Proceedings Graphics Interface 1982*, May, 1982, NCGA (National Computer Graphics Association), Toronto Ontario, Canada, 323-328.

[9] B. Singh, J. C. Beatty, K. S. Booth, and R. Ryman, A graphics Editor for Benesh Movement Notation, CS-82-41, University of Waterloo Computer Graphics Lab, Waterloo, December, 1982, 120 pages.

[10] L. A. Price and C. A. Cordova, Use of Mouse Buttons, *CHI Proceedings*, December, 1983, IEEE, 262-266.

[11] B. Shneiderman, Design Issues and Experimental Results for Menu Selection Systems, CS-TR-1303, CAR-TR-26, University Of Maryland, College Park, July, 1983, 37 pages.

[12] M. Pulver, J. Morrison, and K. Szabo, A Graphics Package for Interactive CAD Applications, VLSI Group, University of Waterloo, January, 1985.

[13] J. Ousterhout, Caesar: An Interactive Editor For VLSI Layouts., *VLSI Design*, 4th Qtr, 1981, Palo Alto, California, 34-38.

[14] W. M. Newman and R. F. Sproull, *Principles Of Interactive Computer Graphics 2nd Edition.*, McGraw - Hill, 1979.

[15] J. D. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison Wesley, Reading, Massachusetts, 1982.

[16] M. I. Elmasry, (ed.), *Digital VLSI Systems*, I.E.E.E. Press, 1985.

[17] M. I. Elmasry, Stick-Layout Notation For Bipolar VLSI, *VLSI Design*, March-April, 1983, Palo Alto, California, 65-69.