Combining Graphics and Layout Language in a Single Interactive System

by

Stephen Trimberger

Technical Report #3794

June 9, 1980

Computer Science Department

California Institute of Technology

Pasadena, California 91125

Silicon Structures Project

sponsored by

Burroughs Corporation, Digital Equipment Corporation,

Hewlett-Packard Company, Honeywell Incorporated,

International Business Machines Corporation,

Intel Corporation, Xerox Corporation,

and the National Science Foundation

Combining Graphics and Layout Language in a Single Interactive System

by

Stephen Trimberger

Caltech Computer Science Department, Silicon Structures Project

Xerox Palo Alto Research Center

Abstract

Layout languages provide users with the capability to algorithmically define cells. But the specification language is so non-intuitive that it is impossible to debug a design in that language, one must plot it. Interactive graphics systems, on the other hand, allow the user to debug in the form in which he sees the design, but severely restrict the language he may use to express the graphics. For example, he cannot express loops or conditionals. What is really needed is a single interactive system that combines layout language and graphic modifications to the data. This paper describes just such a system.

Introduction

Two primary methods for generating integrated circuit mask layout data are layout languages and interactive graphics. Each has tasks which it does well and those which it does not. The result is that users of both kinds of systems are dissatisfied.

Current layout languages usually fall into the "plotter driver" category, features are described by a sequence of commands to draw figures at absolute coordinates. More advanced languages, usually embedded in an existing programming language, have all the powerful control structures that such languages provide, such as loops and conditionals. The power gained by the addition of true programming language facilities to the layout language are enormous and provide the designer with the ability to algorithmically define a circuit or a piece thereof. This algorithmic design is not possible with today's interactive graphic design aids.

Unfortunately, languages specify graphic positions in an awkward fashion, by numbers. A user of current layout languages has a separation between the graphics specification and the graphics viewing. Current languages force the user to go through a tedious and time consuming edit-compile-plot cycle. Interpreted languages get rid of the explicit compilation, but have a corresponding lengthy program execution, which achieves the same effect, slowing down the design cycle. Interactive techniques have attempted to get rid of this lengthy cycle, but have been usually aimed only at the graphic form and not at the language form.

When dealing with graphic data, such as integrated circuits, it is necessary to view the data graphically. Often the limiting factor in the speed of design is the time it takes to plot the data. Interactive graphics systems provide "instant plotting", enabling the designer to iterate extremely quickly on the design.

Interactive graphics systems also give a powerful "language" for handling the data. For example, the user may point to the object of his attention or to a desired position, rather than search for certain numbers in a program printout or type numbers in a program oriented system. But interactive graphics systems do not allow graphic objects to be positioned with respect to other objects, except occasionally, in a most rudimentary adjacency manner. Interactive graphics

systems, because of the limitation of the "language" used to specify objects, cannot specify algorithmically defined cells. Positions are given in some absolute coordinate space and are independent of one another. One cannot express conditionals or relative coordinates.

The major disadvantages of each kind of system correspond to the strong points of the other. Language systems are versatile but tedious to use, graphics systems are easy to use, but severely limited in their expressability. Therefore an attractive idea is to combine both representations in one system which allows modification of the IC data in both forms. This is called parameterized graphics by graphics system users and instant plotting by language system users.

This paper deals with the design, implementation and evaluation of the ideas for combining graphical and textual data representations.

Overview of Sam

Sam is the name of a system which combines the two data representations. Sam was written on a personal minicomputer, the key features of which are a high-resolution black and white video monitor used for both graphics and text output, and a "mouse" graphic pointing device for graphic input, as well as a keyboard and facilities for printing and file storage. Sam runs in the Smalltalk environment, an object-oriented system with very powerful programming and debugging aids [Ingalls 1977]. Smalltalk is a virtual memory system with its own memory manager and garbage collector.

Sam provides the user with a two-part viewing window on the display. The left side shows the program view of the design under edit, the right side shows the graphics view of that cell (Figure 1). The user may move the viewing location in either window and may make edits to the data in either window. When the design is changed in either window, the change is reflected immediately in both windows.

The data displayed in the windows are "pictures" of the data structure. The data structure is the base form, the program view and the graphic view are merely different ways of looking at the base form of the data. When either the graphic bitmap form or the program character-string form is needed for display it is

generated from the data structure. When the user makes what appears to be a modification of the data in either window, the commands are translated into calls on procedures in the data structure to carry out the action. The data structure makes the modification and causes both displays to be updated. The two views are kept consistent because they are both refreshed from the same data im memory.

Internally, Sam consists of four major pieces. The first piece is the data structure, which is more than a conventional design automation database, consisting as it does of objects which have both data and code attributes. Two more major pieces are the Graphic Editor and the Program Editor, which display data and convert inputs to commands to the data structure. The fourth piece is a small coordination piece, which holds together the two editors.

Data Structure

The heart of the Sam system is the data structure. The data structure is modelled after the parse tree of a simple programming language (Figure 2). This facilitates the viewing and editing operations. Since a parse tree form is needed when viewing graphically, such a form must be constructed. It is more convenient and simpler to keep the data in this form rather than re-construct it from a character-string or token-string base language form. The program-editor modifications are passed down the parse tree. Editable fields in the program view correspond to nodes in the parse tree form.

The data structure language includes loops, conditionals and variables. Procedure definition in the language provides the cell definition facility for the integrated circuits with the added feature that cells can have parameters passed to them, just like procedures in current programming languages.

The data structure contains eight kinds of entries: Box, Instance, Cell Definition, Loop, If, Assignment, Block and Comment. Each kind of entry is defined as a Smalltalk "Class", which is a construct consisting of some data and some procedures for manipulating that data. Each statement in the data structure is one "instance"of a Class, it has its own data fields, but shares the procedure code with all other objects of its class. The underlined portions of the statements below correspond to the data fields of each class of objects.

There are data commands for graphic primitives:

```
Box.  Layer: Polysilicon.  ll: 8,1  ur: 10,10.
```

Commands for programming language constructs:

```
PLAsize = PLAdrivSize + (minterms * PLAandSize).

If firston
    Then: Box.  Layer: Polysilicon.  ll: 8,1  ur: 10,10.
    Else: Note: Don't connect the switch

For buscount = 1 to bussize do:
    Note: Connect the busses
    Box.  Layer: Metal.  ll: LeftSide,bottom + 10*buscount
    ur: RightSide,bottom + 3 + 10*buscount.
```

Commands for building the cell instantiation hierarchy:

```
Def andPlane (inputs, minterms, code)
    Note:  The stuff for the andplane goes here.

Inst PLAcellpair t11:1, t12:0, t21:0, t22:1, tx:(14*incount),
    ty:-4+(14*mincount) | Params: (code(mincount*2-1, incount)),
    (code(mincount*2, incount)).
```

The Block command allows many statements to be grouped into one for inclusion in a loop, for example. Blocks show indented:

```
    Note: There is nothing in this loop.
    Note: Except these comments.
```

And, of course, a comment:

```
Note: Tricky stuff: Be sure DEI and CLS are never both high...
```

The procedures recognized by the data objects define the interface to the data structure. In particular, each class has procedures for updating each of its data fields. The fields of a Box are the layer and the x-y positions of thecorners of the box. An If statement, on the other hand, has three fields, the conditional

expression, and two pointers to other Sam statements, one for the THEN-branch, and one for the ELSE-branch.

The Classes have procedures to show graphically and print textually in the respective windows. These two procedures provide the pictures of the data structure that the user sees when he manipulates the data. Commands from the two editors, one textual, one graphical, to alter the data, are translated into calls to statement instances in the data structure to change a certain field. These calls may be passed down the tree if necessary. Thus, there is a common interface for both representations.

Program Editing

Sam provides a syntax-directed editor for the program view. This is similar in philosophy to interactive graphics editors. The user may not alter arbitrary pieces of the picture of the data, be it individual bits in the raster of the graphics or, in this case, individual characters in the text. Instead, the user may only manipulate complete syntactic pieces of the data, such as whole Boxes or complete expressions.

Therefore, the syntax of the program view need never be checked. It is always correct because it is impossible to make it incorrect. The editing features do not allow the "o" to be deleted from the "For" keyword, for example. Only meaningful pieces of the data can be changed.

Examples of complete syntactic objects are whole statements, expressions and numbers. These correspond exactly to the underlined portions of the statements shown above. In the graphic editor, the user may stretch, move create and destroy Boxes. These also can be seen as changes in the expressions that make up the position of the Box and changes in the Blocks that contain the Box statements (for create and destroy). When editing an expression, a variable name or the comment text in a Note statement, the user modifies the actual text, which is re-compiled when an attempt is made to terminate the edit. This gives full generality and ease of expression when editing at the lowest level.

The changes to the pieces of the program are translated into calls on the procedures of the data objects to effect a change in the data structure. When a data object is

changed, both pictures of the data are immediately updated to reflect the new data structure.

Updating Problems

There are problems that arise in a system of this sort where changes can be made in two different forms and which must remain consistent. There are two problems of particular importance because of their frequency: expression update and loop update.

Expressions. Suppose the x-position of a Box is given by the equation "3*w+4" and suppose further that the Box was moved graphically. How should the x-position be represented now?

Let us make this an example. Assume w=2. "3*w+4" is 10. In the graphics window, the user sees the x-position as 10 and moves it to 13. The resulting expression could be any of the following expressions which evaluate to 13:

`13`	destroy the parameterization
`3*w+7`	add a constant, translating the position
`(13/10)*(3*w+4)`	multiply by a constant, scaling the position
`3*w+4 (w=3)`	change the value of the identifier

The first choice, the most simple, destroys the parameterization. The parameterization may still be relevant and, in any case, is useful to the user in understanding the design, so this may not be very wise. The second and third choices preserve the parameterization, but there is no assurance that this is what the user wanted, either. The last solution is fairly tricky. Since w coulditself be defined as an expression, we are faced with this same problem again when updating w. The result is a constraint satisfaction problem. Small changes in the design could have far-reaching and non-obvious effects on the circuit.

None of the solutions can give the correct result every time. The program cannot know the mind of the user. One option is to give the user several different graphic editing modes, one for each of the choices above. This leads to terrifically cluttered user interface and much chance for error if the user accidentally modifies

something with the wrong mode. Another solution could be used where expressions of the form "aX+b" are translated, because the variable is already translated; expressions of the form "aX" are scaled because the variable is already scaled and of the form "X" modify the variable. Or the system could translate all positions and scale all dimensions. But these guesses could still be wrong, and the user would have to remember all the special conditions. In general, a blatantly naive, but consistent system is better than a clever, but inconsistent one.

Sam translates all changes. This keeps the modifications local and preserves parameterization. In use, I found that this was exactly what I wanted I every case, I wanted graphic features translated with the parameterization preserved.

Loops. When one graphically edits the graphics corresponding to one iteration of a loop, should all the iterations be changed, or just the one? Typically, language systems modify all iterations (changing the object in a step and repeat) while graphic systems either do the same or disallow the operation. Sam modifies all iterations of the loop. This seems to work well, but there are clear cases where one is preferred to the other. This may be a situation in which two different editing modes would work. This has not yet been fully investigated.

General Evaluation of Sam

The individual editors used in Sam were made intentionally weak in order to simplify the programming task so that the project could be completed quickly. These weaknesses were easy to identify and ignore when evaluating the new ideas in Sam and they will not be discussed here. Instead, this section will cover problems arising from combining these two data representations as described in the preceeding sections.

The language model used for Sam's data structure was inadequate. There were two big problems with the language, one anticipated, one unexpected. Sam's data structure language was modelled after a very simple Algol-like language without data scoping or type checking. This made the interpreter simple and obviated the need for error messages. While the control structures were adequate, the data structures were not. More advanced data types such as points and rectangles as well as arrays were needed. I was able to bypass these problems in the evaluation, but a

real system would have to provide them. Also, there seemed to be no end to the data types needed. A complete structured data facility is needed.

There was a major problem with incremental data updates. Either too much or too little of the data was re-interpreted when updating the pictures. This was because the language model did not provide a facility for expressing dependence of statements. (A Box statement with a variable in one of its expressions depends on the assignment statement that sets the value of that variable). In languages, independence is expressed by concurrency. Two pieces of program can run concurrently if they are independent. Therefore, a proper language model for Sam would have to be able to express concurrency. Concurrency can be implemented in a number of ways, using COBEGIN-COEND as in Concurrent Pascal [Brinch-Hansen 1973], FORK-JOIN primitives or an implied-concurrent language or single assignment language. At present it is not known which is better. The choice should be made from the point of view of the data structure, not the user, as the user of the system will never need to deal with the concurrent aspects of the language -- that can be handled totally internally.

Cell definition was handled badly in Sam. A cell definition was implemented as another statement in the language. This caused unnecessary clutter and clumsiness in the language view. Cells should be treated as separate pieces of design, in their own separate coordinate space. By treating them as statements, much of the independence of design was lost.

Sam's cells were very good in that they were parameterized. The power of parameterization cannot be overemphasized. When placing an instance of the cell, one could supply parameters to alter the internal structure of the cell as desired. This is the same as passing parameters to a procedure in a programming language, and is done for the same reason. It allows the cell to be used in many more situations. However, Sam's cells did not give information back to the user in the program picture. For example, a cell should have connection points on it, which could be used in the program to connect wires. To do this, instances of cells must be named. Attributes of Boxes should be accessible, so they Boxes should be named also. This implies that Instances and Boxes should resemble SIMULA class instances in the program view, exhibiting attributes.