Combining Graphics and a Layout Language in a Single Interactive System

by

Stephen Trimberger

Technical Report #4281

May 1981

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 A LAYOUT LANGUAGE IN A SINGLE INTERACTIVE SYSTEM

Stephen Trimberger

California Institute of Technology, Computer Science Department, Silicon Structures Project
Pasadena, California

XEROX Palo Alto Research Center
Palo Alto, California

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 *interactive graphics* and *layout languages*. 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.

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 provide 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. Positions are given in some absolute coordinate space and are independent of one another. Many systems give the ability to replicate a piece of geometry. This is a looping construct, but it is severely limited by the capabilities of the graphics system. Graphics systems do not allow the expression of conditional geometry or relative positioning. Much more powerful language-style operations are needed.

Layout languages attempt to resolve these problems. Layout languages usually fall into the "plotter driver" category. Features are described by a sequence of commands to draw geometry 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 provides the designer with the ability to *algorithmically define* a circuit or a piece thereof.

Algorithmic definition is the specification of a piece of a layout with an algorithm. The algorithm that generates the layout can be parameterized, giving the ability to define, for example, an *n*-input NAND gate, a line driver with exactly the required power, a Programmed Logic Array (PLA), or even an *n*-bit processor. Such cells are much more versatile than typical "hard" standard cells. This algorithmic design is not possible with current interactive graphic design aids.

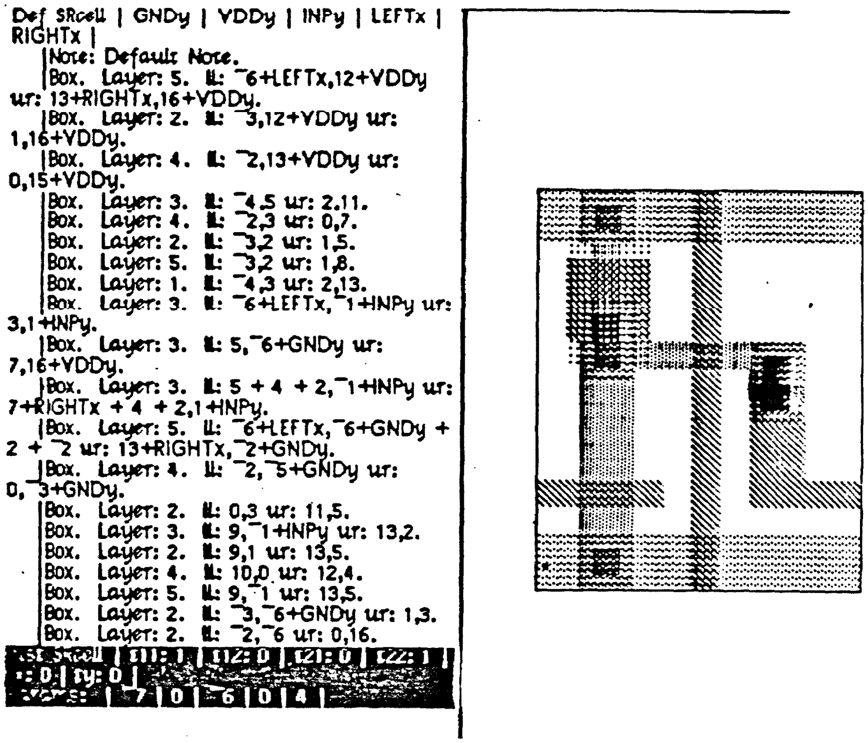
Unfortunately, languages specify graphic positions in an awkward fashion, by numbers. A user of a layout language system has a separation between the graphics specification and the graphics viewing. Current languages force the user to go through a tedious and time consuming edit-compile-plot cycle. Interpreted languages get rid of the explicit compilation, but have a corresponding lengthy program execution and evaluation cycle, which achieves the same effect, that of 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.

The major disadvantages of each kind of system correspond to the strong points of the other. Graphics systems are easy to use, but severely limited in their expressability, language systems are versatile but tedious to use. Therefore an attractive idea is to combine both representations in one system which allows modification of the integrated circuit 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. The work on Sam was done at the Xerox Palo Alto Research Center. Sam was written on the *Alto*, 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" 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



```
Def SRcell | GNDy | VDDy | INPy | LEFTx |
RIGHTx |
   |Note: Default Note.
   |Box.  Layer: 5.  ll: ¯6+LEFTx,12+VDDy
ur: 13+RIGHTx,16+VDDy.
   |Box.  Layer: 2.  ll: ¯3,12+VDDy ur:
1,16+VDDy.
   |Box.  Layer: 4.  ll: ¯2,13+VDDy ur:
0,15+VDDy.
   |Box.  Layer: 3.  ll: ¯4,5 ur: 2,11.
   |Box.  Layer: 4.  ll: ¯2,3 ur: 0,7.
   |Box.  Layer: 2.  ll: ¯3,2 ur: 1,5.
   |Box.  Layer: 5.  ll: ¯3,2 ur: 1,8.
   |Box.  Layer: 1.  ll: ¯4,3 ur: 2,13.
   |Box.  Layer: 3.  ll: ¯6+LEFTx,¯1+INPy ur:
3,1+INPy.
   |Box.  Layer: 3.  ll: 5,¯6+GNDy ur:
7,16+VDDy.
   |Box.  Layer: 3.  ll: 5 + 4  + 2,¯1+INPy ur:
7+RIGHTx + 4  + 2,1+INPy.
   |Box.  Layer: 5.  ll: ¯6+LEFTx,¯6+GNDy +
2 + ¯2 ur: 13+RIGHTx,¯2+GNDy.
   |Box.  Layer: 4.  ll: ¯2,¯5+GNDy ur:
0,¯3+GNDy.
   |Box.  Layer: 2.  ll: 0,3 ur: 11,5.
   |Box.  Layer: 3.  ll: 9,¯1+INPy ur: 13,2.
   |Box.  Layer: 2.  ll: 9,1 ur: 13,5.
   |Box.  Layer: 4.  ll: 10,0 ur: 12,4.
   |Box.  Layer: 5.  ll: 9,¯1 ur: 13,5.
   |Box.  Layer: 2.  ll: ¯3,¯6+GNDy ur: 1,3.
   |Box.  Layer: 2.  ll: ¯2,¯6 ur: 0,16.
Set SRcell | t11: 1 | t12: 0 | t21: 0 | t22: 1 |
tx: 0 | ty: 0 |
Vars: | ¯7 | 0 | ¯6 | 0 | 4 |
```

Figure 1. Snapshot of the Sam Display.

manager and garbage collector.

Sam was meant to be an experiment, a quick implementation to test the ideas for combining graphic and language systems. For this reason, many decisions were made to facilitate the implementation at the cost of execution speed and fullness of the user interface. Smalltalk was chosen as the implementation language because of its virtual memory, automatic garbage collection, excellent debugging facilities and powerful language constructs which facilitated code sharing. These features were necessary in order to finish the system in the three months allotted for the project. The price of these features was the inability to perform low-level hacking to improve performance. Since no such hacking was to be done, there was essentially no cost for the powerful language environment.

Sam provides the user with a two-part viewing window on the display as seen in figure 1. The left side shows the program view of the design under edit, the right side shows the graphics view. 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 in memory.

Internally, Sam consists of four major pieces (see figure 2). 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 as seen in figure 3. Each node in the data structure corresponds to one statement in the simple programming language. For this reason, I use the phrase *data structure language* when referring to the operation of the data structure. The parse tree form facilitates the viewing and editing operations. It is more convenient and faster to keep the data in this form than it is to re-construct it from a character-string or token-string base language form when it is needed for graphic operations.

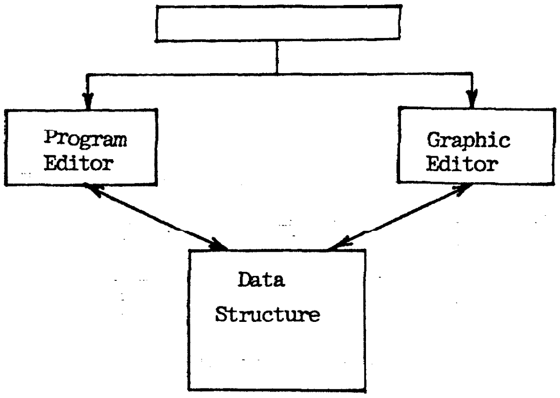


Figure 2. Sam Block Structure.

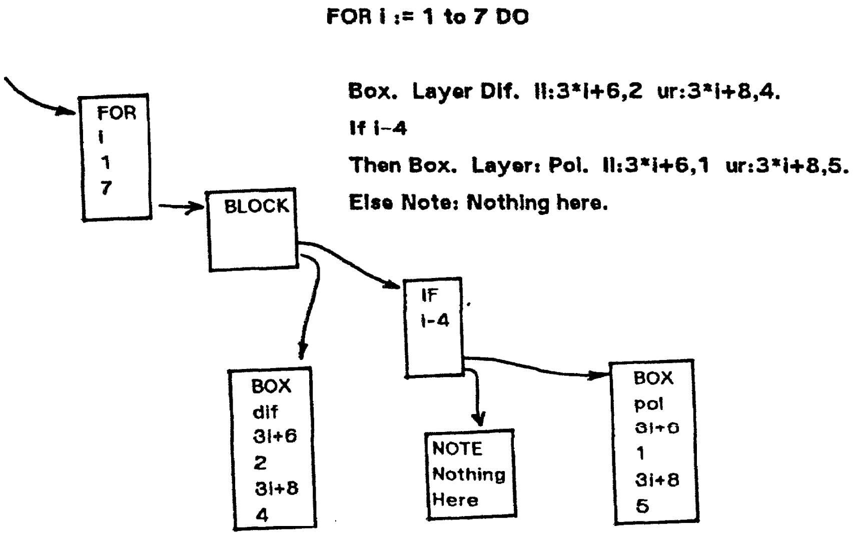
The data structure language includes loops, conditionals, and variables. Procedure definition in the language provides the cell definition facility for the integrated circuits. Thus, cells defined in Sam can have parameters passed to them, just like procedures in programming languages.

The data structure contains eight kinds of entries: *Box*, *Instance*, *Cell Definition*, *Loop*, *If*, *Assignment*, *Block* and *Note*. Besides the major data structure entries, there are entries for *name*, *expression*, and *comment text*. 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, which has its own data fields, but shares the procedure code with all other objects of its class.

The statements below are the text representations of the data structure entries. These correspond to the text view of the design. The underlined portions of the statements correspond to the data fields of each class of objects. The data fields are the portions of each object which may be manipulated by the editors. Commands from the editors to modify the design are translated into commands to change one or more of the underlined fields in a data structure entry.

The *Box* entry is the graphic primitive, and is described by a layer, and expressions for the lower left and upper right x-y positions of the corners of the rectangle.

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



FOR i := 1 to 7 DO

Box. Layer Dif. ll:3*i+6,2 ur:3*i+8,4.
If i-4
Then Box. Layer: Pol. ll:3*i+6,1 ur:3*i+8,5.
Else Note: Nothing here.

Figure 3. The Sam Data Structure.

The data structure contains entries for programming language constructs. The first is an assignment entry with a destination variable name and source expression. There is a conditional entry with three fields: the conditional expression and two pointers to other Sam data structure entries, one for the *THEN*-branch and one for the *ELSE*-branch. Sam has an entry for iteration, which consists of loop variable name, starting and ending loop value expressions, and the entry for the body of a loop. The *Note* entry is a comment, and is used for annotation.

```
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.

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

There are entries for building the cell instantiation hierarchy. The cell definition entry has a cell name, a list of parameter names and an entry for the body of the cell. The cell instance entry has the name of the cell to be instantiated, the transformation matrix entries to specify the position and orientation of the instance, and a list of parameters.

```
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* entry 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.
```

The procedures recognized by the data objects define the interface to the data structure. In particular, each class has procedures to update each of its data fields shown underlined above. In addition, each class has 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 entries in the data structure to change a certain field, giving a common interface for both representations. The calls may be passed down the tree if necessary. An operation on an *If.* statement may be one on the statement itself, in the case of modifications to the conditional expression, or may be passed down the *THEN*-branch or *ELSE*-branch, in the case of a textual *select* operation.

EDITING THE DESIGN

Sam provides a *syntax-directed editor* for the program view. This is similar in philosophy to interactive graphics editors, since 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. Complete syntactic objects are whole statements, expressions, and names. These are, by definition, the data structure entries, shown underlined in the list of data structure entries, above.

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. 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 program editor allows the user to *select* a statement or subfield of a statement by pointing to it. When this is done, the selected entry shows video inverted in the program window, and outlined in the graphic window. Selected items may be deleted or modified by commands to either editor. The program editor has commands to create any statement, delete the selected statement, move and copy textually, and edit expressions, names, and comment text.

The graphic editor commands are very similar to those available in commercial systems. In the graphic editor, the user may select a box by pointing to it. The selection works exactly the same in the graphic editor as it does in the program editor. The user may manipulate boxes with commands to create, destroy, stretch, and move and copy graphically. The graphic modifications are interpreted as changes in the expressions that make up the position of the *Box*, for modifications of existing objects, and as changes in the *Block* that contains the *Box* statements, for creating and destroying elements.

The changes from both editors to entries in the data structure are translated into calls on the procedures of the data objects to effect the change. When a data entry 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 which must remain consistent. There are two problems of particular importance because of their frequency: expression update and iteration 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 (see figure 4). The resulting expression could be any of the following expressions which evaluate to 13:

13	destroy the parameterization
3*w+7	add a constant (translate)
(13/10)*(3*w+4)	multiply by a constant (scale)
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* could itself 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 a cluttered user interface, increasing the chance for subtle errors 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 expression already has a translation; expressions of the form "aX" are scaled because the variable is already scaled and expressions 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 effects of modifications localized and preserves the parameterization. In use, this was found to be the proper choice in every case. It seems to be a reasonable solution, preserving parameterization in a simple, straightforward manner.

Iteration: 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 of 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 when the other choice is preferred. This may be a situation in which two different editing modes would work. The iteration problem 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 covers problems arising from combining these two data representations.

The evaluation of Sam consisted of the design of cells of varying complexity: an inverter, a totally graphical task; a simple, parameterized *stretchable* shift register cell, which could change its pitch depending on input parameters; and a PLA, a predominantly algorithmic task. The interaction of the two data representations indicated that even the simple Sam system was unusually powerful. The details of the evaluation will not be discussed, but the results are reported here.

Box. Layer: Pol. ll:3*w+4,6 ur: 3*w+12,9.

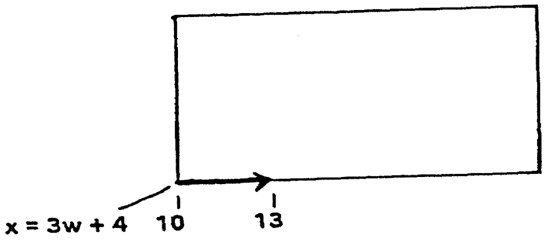


Figure 4. Expression Update.

With the larger designs, Sam swamped the minicomputer on which it ran. The problem was due mostly to the limited memory space. Since Sam was meant to be a quick and dirty experiment, no coding tricks were used to speed the execution. Sam ran acceptably fast on a more powerful processor with a larger address space.

The language model used for Sam's data structure was inadequate. There were two major problems with the language. First, the Sam 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. However, structured data types such as points, rectangles and arrays were needed. This need was anticipated and the problems were bypassed in the evaluation, but a real system would have to address these data structure issues.

The Sam data structure language could not properly handle 'incremental data updates 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 programming languages, independence is expressed by concurrency, since two pieces of program can run concurrently if they are independent. A piece of the design must be refreshed only when it is dependent on something which has changed. Therefore, a proper language model for the Sam data structure would have to be able to express concurrency. This concurrency would never be seen by the user, since is only used by the system internally.

One deficiency of the Sam data language was not a problem. Since loop termination could not be affected inside the loop, the language is equivalent to a finite state machine, and less powerful than a Turing machine. During the evaluation of Sam, this limitation was not a problem. This would seem to imply that a finite language is sufficient to describe a finite object, such as an integrated circuit chip. This question is still unresolved, however. Some evidence exists that data-dependent recursion or iteration is necessary to produce some designs.

The power of parameterization in Sam's cells was very good. When placing an instance of a 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 procedure/cell to be used in many more situations.

A cell should have connection points on it, which could be used as variables in expressions. These could be used in the program view to connect wires. Attributes of *Boxes* should be accessible also, for the same reasons. This implies that *Instances* and *Boxes* should exhibit attributes in the program view, like SIMULA class instances [Birtwistle 1973].

One graphic editing feature that would have simplified many operations is one which would position new features relative to a point. Then, all items relative to that point could be moved just by moving the point. This would allow huge pieces of the design to be moved quickly and easily by parameters in the program.

Perhaps the most powerful single feature of Sam is the selection operation. The selection shows the relationship between program statements and graphic elements, enabling the designer to move quickly and easily between representations. Not only does Sam give an instant plot, but the plot is an active entity, telling the user where each graphic feature comes from in the program.

CONCLUSIONS

Sam is fundamentally different than any commercially available design system. The Sam system gives the user the ability to pass quickly between the graphics and language worlds. The mating of graphics and language in this fashion provides the user with a vast increase in expressive power without losing the fast iteration of graphic systems. The Sam system was developed in a deceptively short amount of time. A full, usable Sam system requires a good graphics system and a good language system as well as facilities relating the two.

Complex designs can be created algorithmically without giving up the rapid feedback of graphics systems. Such a system can be used to generate a parameterized cell library, which could be used to design a large number of chips. The cells in the Sam library could be parameterized in such a fashion that calls to them would provide a behavioral design language. Thus, large designs could be created from a behavioral description, and the layout could still be optimized graphically. Work is progressing along both these paths.

REFERENCES

[Birtwistle 1973] G. Birtwistle, O.J. Dahl, et. al., *Simula Begin*, Petrocelli/Charter, 1973

[Ingalls 1977] D. Ingalls, "The Smalltalk-76 Programming System Design and Implementation", Proceedings of the Fifth Annual Symposium on Principles of Programming Languages, January, 1977