Automated Standard Cell Synthesis using Genetic Algorithms

Anil Bahuman
Artificial Intelligence Center
University of Georgia
Athens, GA 30602
abahuman@ai.uga.edu

Benjamin Bishop
Dept. of Computer Science
University of Georgia
Athens, GA 30602
bishop@cs.uga.edu

Khaled Rasheed
Dept. of Computer Science
University of Georgia
Athens, GA 30602
khaled@cs.uga.edu

Abstract

We have demonstrated the possibility of applying a modified genetic algorithm (GADO) to completely automate the design of a cell given only a behavioral description and optimization criteria such as power, area, speed or their combination. GADO maintains a population of potential standard cell designs. The designs are evaluated by (1) internal rules (2) MAGIC for design rule checks and (3) SPICE for correctness. Connections between transistors are encouraged by an influence check, which checks for broken connections and floating I/O nodes. Working inverters with arbitrary label placements are designed by evolution as a proof of concept.

I. INTRODUCTION

Mazumder and Rudnick [12] provide a good survey of genetic algorithms as a tool to solve design automation problems faced by the VLSI community and demonstrate some genetic algorithm based solutions. The problems considered include partitioning, automatic placement, automatic routing, automatic test generation, power estimation and technology mapping for FPGAs
We demonstrate the possibility [3] of one such tool that can assist engineers to build standard cells on the fly customized to ad-hoc constraints that they need to address in their design. Our method is an example of design by evolution using genetic algorithms.
Standard cell methodology is widely used in IC design. Considerable effort [1,11,12,15,16] has been invested in attempting to automate the design of a standard cell. Automation of standard cell layout generation (1) enables rapid design of cells with [near] hand design quality, (2) provides a test bed for evaluating new process technologies by allowing rapid synthesis of cells and (3) enables rapid migration of designs to new process technologies.
Most methods for cell synthesis are schematic-based methods as they start with a sized/unsized transistor-level netlists (specifying how the transistors are connected) and automate the processes of placement, followed by routing and possibly compaction. Our implementation combines all these three steps into a single phase and attempts to evolve layouts on the fly given only a behavioral (truth-table like) description.

The input specification consists of: (labels and cell templates are defined in Section II B)

1. A truth table description of the input and output with as many possible transitions (argued in Section I B).
2. A set of building blocks consisting of geometries of objects such as transistors, piece of polysilicon, piece of polycontact etc (Figure 2).
3. A classification of labels used (as inputs and outputs).
4. A cell template specifying the placement of fixed ports (if any).

MAGIC [17] (a layout editor) and SPICE [18] (a circuit simulator) are also used by the genetic algorithm to evaluate and validate designs. The genetic algorithm uses these external simulators to evaluate its designs and attach the final fitness value to a candidate design (Figure 10). MAGIC is used to check for design rule errors and SPICE is used for simulating the circuit and checking for correctness in the circuit behavior.

A. Motivation

As noted earlier there is a progression toward more automated cell synthesis techniques. Some authors [5,6] advocate a shift in the use of standard cell libraries, as we know them. The idea here is to engineer standard cells on demand or "on the fly" thus eliminating the need for developing and maintaining expensive static cell libraries. This would allow the designer to demand cells that may not exist in any pre-defined library, optimized for a given criterion depending on the current constraints. In particular this would permit [10]:
1. Standard cell and datapath placement and routing tools to request cells with exact I/O placement.
2. Interconnect optimization tools to request cells with specific input and output impedance values.
3. Power optimization tools to request specific power/delay trade-offs, perhaps even specify a logic family.

Other advantages include:
1. Synthesis of non-rectilinear cells of "optimal" dimensions.
2. Transistor level tuning (choosing, ordering, sizing and placing).
3. Rapid migration of designs to new process technologies.

Typical techniques are schematic dependent. They either start from a netlist of transistors with the transistor type and sizing specified by the designer or from a symbolic layout followed by compaction. Existing compaction techniques tend no to change the shape or orientation of layout objects and thus depend on the designer's specification. Yet another technique used is the development of procedural module generators that are programs that hard code procedures for assembling a cell as well as design rule constraints. Hard coding limits their use in the face of changes in technology [5].

Fixed libraries make device level tuning impossible (or difficult). Poor timing, area or power tradeoffs may result since the design of the cell is decoupled from the constraints imposed by cell placement tools. Schematic independence and device-size tuning is best accomplished via on the fly leaf cell synthesis. The use of C5M to develop a 440 MHz processor for IBM discussed in [6] is a good example of such a design methodology.

Lefebvre, Marple and Sechen [5] note that layout synthesis tools need to optimize across all of the following phases since each of the phases involve design tradeoffs:

1. Creation of a transistor circuit topology that provides a certain digital function.
2. Sizing and ordering transistors in the circuit topology.
3. Placing routing and compacting the above transistors into a layout.

B. *Our Research Goal*

Given the goal of automating phases 1, 2 and 3 above into a single phase, we set out to explore the possibility of accomplishing this for a very simple circuit – a CMOS inverter – without a schematic and without incorporating any design heuristics for transistor sizing, splitting, ordering, placement, routing or compaction.

Our goal was to start from a behavioral description for an inverter, an optimization criterion and a set of building blocks (different types/clusters of transistors, piece of “poly”, piece of “metal1”, piece of “ndcontact” etc.) and attempt to automate the process of finding a working inverter optimized for the given criterion as a proof of concept.

The behavioral description consisted of a truth table of inputs and outputs with all possible input transitions (0 to 1 and 1 to 0). This is because some exotic logic families may fail on certain input transitions. Table 1 lists all the input transitions used for the inverter with corresponding outputs.

TABLE 1: BEHAVIORAL DESCRIPTION

INPUT	OUTPUT
0	1
1	0
0	1

But what about testing different input sequences? There are an infinite number of input sequences (varying length) and it is possible that a design with extra state information fails beyond out test sequence. Our own experience with this algorithm is that it tends to find the simplest design and hence the correct design i.e. minimal state information. This is because complicated designs that have more components also occupy more area and hence receive higher penalties than simple designs (i.e. minimum state information). (The penalty scheme used in the fitness function is discussed in 2.3).

Another question that needs to be addressed is the justification for using a behavioral description as opposed to the traditional schematic description. We believe that this added degree of freedom creates more room for innovation. The algorithm simultaneously searches for the optimal layout as well as the optimal schematic.

II. AN EVOLUTIONARY APPROACH

A. *Genetic Algorithms for Design Optimization*

Genetic Algorithms (GAs) are a class of heuristic algorithms that can search for a solution by a process similar to natural selection i.e. the desired characteristics of the solution are enhanced by continued breeding [14]. GAs attempt to find good solutions by randomly creating a pool of potential solutions to the problem and manipulating those solutions by the use of *genetic operators*. Genetic operators manipulate existing solutions to generate new solutions. Each solution is assigned a fitness value which is a numerical assessment of how well it solves the problem.

We use GADO [4], which is a steady state *Genetic Algorithm for Design Optimization* tool developed by Khaled Rasheed, for applications in engineering design optimization where search spaces are very difficult to search and evaluating fitness using external simulators can be quite time consuming.

GADO maintains a population of potential designs (standard cell designs in our case). New designs are created by combining past designs using crossover and mutation operators.

A steady state GA model is used, in which the operators are applied to two parents selected from the elements of the population via some selection scheme, one offspring point is produced, then an existing point in the population is replaced by the newly generated point via some replacement strategy. Here selection was performed by rank (the probability of being selected depends on the fitness rank in the population rather than the actual fitness value) because of the wide range of fitness values. The replacement strategy used here is a crowding technique [20], which takes into consideration both the fitness and the proximity of the points in the GA population. The GA stops when either the maximum number of evaluations has been exhausted or the population loses diversity and practically converges to a single point in the search space.

Several crossover and mutation operators are used [4], most of which were designed specifically for the target domain type. GADO also uses a search-control method [4] that saves time by avoiding the evaluation of points that are unlikely to correspond to good designs.

B. Representation

The representation refers to encoding a layout into a series of numbers that form an individual "string" (Figure 1). We use integers to encode a layout (Table 2 specifies the bounds). Each design is represented in the algorithm as a string of Objects corresponding to each component in the physical layout:

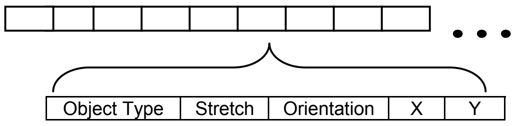

Figure 1: Coding Scheme

An object is a member of a pre-defined set of building blocks including various types of transistors, piece of polysilicon, piece of ploycontact etc. Figure 2 is a snapshot of a MAGIC file with all the blocks placed in a single file.
It may be noted that there are 15 object types that consist of symmetrical and asymmetrical transistors of different types (single, dual and triple). Three other blocks are pieces of polycontact, polysilicon and "metal1" (extreme right, top to bottom in Figure 2). The user may easily add other building blocks as and when required.

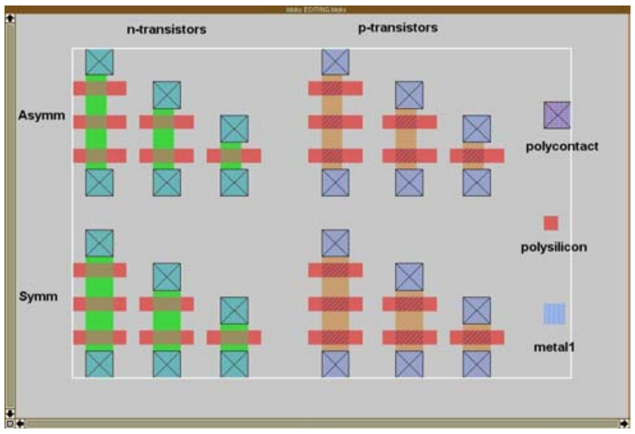

Figure 2: Building Blocks

Each object is further defined by its type, X and Y coordinates, stretch and orientation (Figure 1). Cell limits refer to the size of the grid of the maximum allowable cell size (as defined by the user). This forces the GA to place components only within the grid.
The orientation refers to a number (0, 1, 2 or 3) that decides the rotation applied to a block (0, 180, 90 or 270 degrees respectively).
The stretch factor is a number corresponding to the number of units by which a block is scaled along the direction of orientation. The scaling for transistors is done in way so as to stretch the middle region only thus keeping the length of the polysilicon overhangs constant. The maximum scale is limited by the maximum allowable cell size
X and Y refer to the coordinates of the center of the object and are thus also constrained to be within the maximum allowable cell area.

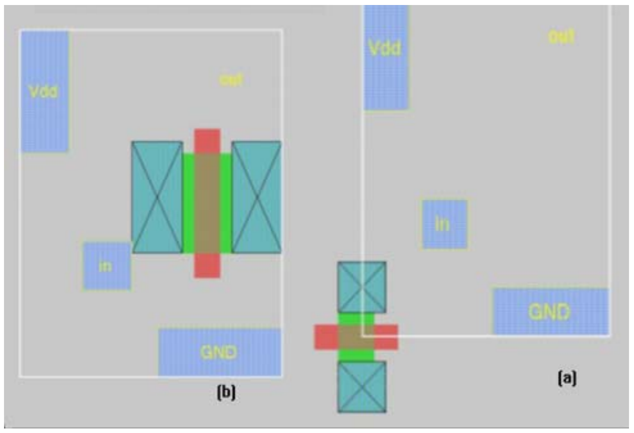

Figure 3: Example Cells. (b) shows the same object stretched and turned

TABLE 2 PARAMETERS IN THE CODING SCHEME

PARAMETER	VALUE	Fig 3 (a)	Fig 3 (b)
Object Type	1-15	11	11
Orientation	0-3	0	2
Stretch Factor	Cell limits	0	5
X	Cell limits	0	15
Y	Cell limits	0	14

Table 2 summarizes the parameters of each object (column 1), the range of values that they may have (column 2) and illustrates the values corresponding to the objects shown in figures 3 (a) and (b). Note the effects of moving, stretching and rotating the same object. Figure 3 also demonstrates the use of templates. The 3 pieces of metal and the 4 labels (Vdd, Gnd, in and out) are part of a user-defined template over which the other objects are superimposed.
A repair method was used to reduce the effects of aliasing. The objects in the string (chromosome) are sorted before crossover and mutation in a way that maps the order of objects in the cell.

C. Fitness Function

The fitness function checks a candidate design against a list of constraints and penalizes it for every violation encountered. The penalty values vary for each constraint depending on their importance. This way the GA fixes the most serious violations first and progresses to make smaller fixes. The stages in the evolution of a circuit are thus infeasible, incorrect, correct and finally optimized.

In the first phase the GA tries to weed out bad designs such as those with overlapping transistors. After succeeding it goes on to check *connectivity* of individual transistors and tries to encourage connections (more details ahead).

Simultaneously, the design rule checker is invoked so that the GA tries to minimize both design rule violations as well as broken connections.

After the GA has succeeded in creating designs with no broken connections it passes on to the next phase where the design rule check is made in conjunction with a circuit simulation. In this stage, the GA is trying to simultaneously satisfy the requirements of zero design errors and zero simulation errors.

Once the GA has come up with a working circuit it proceeds to optimize the design for a given criterion (i.e. search in the space of correct designs), which could be minimizing area, delay or power dissipation (or some combination of the 3).

One of the most challenging aspects of the project was developing the *influence check* module (part of the fitness module in Figure 10) that checks transistor connectivity and discourages broken connections.

A directed graph was used to capture connectivity information. To illustrate the concept of nodes and connections here is an example of a circuit layout abutted by the graph used to capture connectivity information.

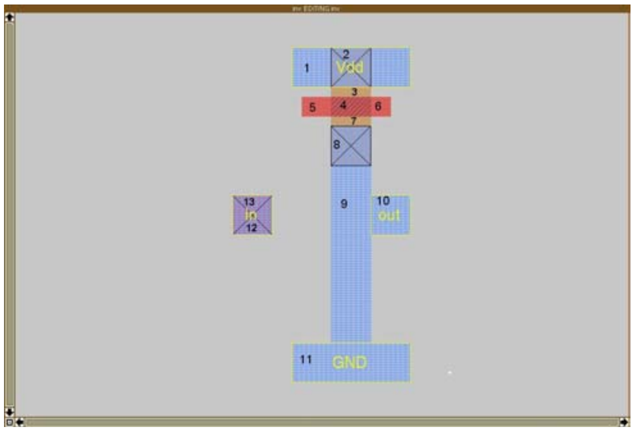

Figure 4a: An arbitrary cell

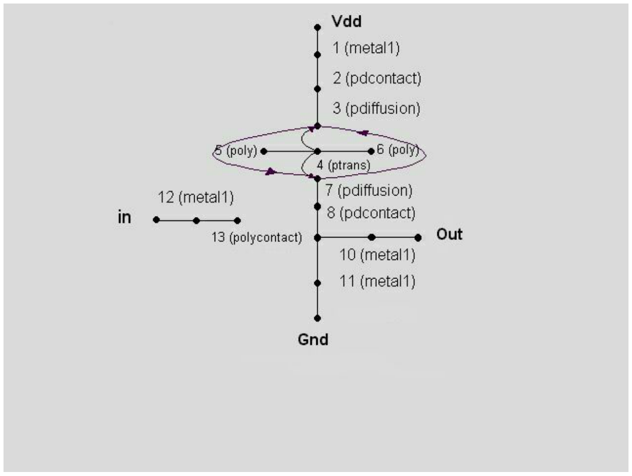

Figure 4b: Connectivity graph of the cell in Figure 4a

The graph is used to penalize nodes that do not influence nodes that they ought to.

We also need to formally define *influence* (arrows in the graph). Node A influences node B if there is (1) a direct electrical connection, or (2) a uni-directional electrical influence such as the control of the gate over the source and drain of a transistor. In Figure 5, nodes 1 and 2 influence one another by property (1); the bi-directional nature of influence in represented by solid lines. Also nodes 3 and 4 exhibit property (2); an arrow going from node 4 to node 3 represents the unidirectional nature of influence.

It may be also noted that all labels were classified as inputs or outputs. Vdd and Gnd were treated as inputs.

The rules that the *influence check* tests are:

1. None of the labels should be shorted
2. Every input must influence at least one output
3. Every output must be influenced by at least one input
4. Every transistor gate must be influenced by at least one input
5. Both source/drain terminals of a transistor must be influenced by at least one input or influence at least one output

Thus the layout in Figure 4a violates rules 2 and 4 above as may be seen in the corresponding graph in Figure 4b.

Let us consider another case where there is a violation, say in rule 3 above (not shown in figures). It is more useful to know how bad the violation is than a binary yes/no.

This is accomplished by a distance check function that in case of a violation in rule 3 will perform a graph traversal starting from the floating output to determine all the nodes that may influence it creating an Output-list.

Similarly another graph traversal is made from each input to create their respective Input-list of nodes that they influence. Once created, the distance function finds the shortest Manhattan distance between nodes in the Input-lists and the nodes in the Output-list and returns a penalty based on this distance. Note that many of these nodes may be in different layers. We use a look up table (Table 3) with preset values that are used to calculate distances between nodes in different layers.

Since designs in which a node in the Input-lists is closer to a node in the Output-lists receive lower penalties, the GA favors these designs and in conjunction with the other 4 rules above it has the holistic effect of encouraging connections between inputs and outputs via transistors without shorting inputs.

This idea lies at the heart of the GA's success. To summarize:

If you have some terminal that is not being influenced by any other terminal, we want to know how close it is to some terminal that can influence it.

(*terminals* include labels and transistor terminals)

TABLE 3: DISTANCES BETWEEN CMOS LAYERS (TOP ROW REPRESENTS LAYERS AS NUMBERED IN THE FIRST COLUMN)

No →	Poly	Ndiff	Pdiff	M1	M2	Ntran	Ptran	PC	NDC	PDC	M2C
Poly	0	1	1	1	2	0	0	1	2	2	1
Ndiff	3	0	3	1	2	4	4	2	2	0	1
Pdiff	3	3	0	1	2	4	4	2	2	0	1
M1	1	1	1	0	1	2	2	0	0	0	0
M2	2	2	2	1	0	3	3	1	1	1	0
Ntran	0	0	2	2	3	0	1	1	1	3	2
Ptran	0	2	0	2	3	1	0	1	3	1	2
PC	1	2	2	0	1	1	1	0	1	1	0
NDC	2	0	2	0	1	3	3	1	0	1	0
PDC	2	2	0	0	1	3	3	1	1	0	0
M2C	1	1	1	0	0	2	2	0	0	0	0

The idea that information about the degree of the violation is more useful than just knowing whether there is a violation is another key ingredient to the GA's success and is used in the next 2 stages too. In the design rule check stage, a script is fed to MAGIC (in NULL mode) and the number of tiles (MAGIC draws error tiles to show regions violating design rule errors; more tiles usually imply more errors) that are involved in the design rule errors is summed to arrive at a penalty.

In the circuit simulation step, a SPICE simulation is made and a penalty is arrived at by dividing the total number of entries in the truth table simulated incorrectly divided by the total number of entries in the truth table.

D. *Smart Diversity Function*

A distance function takes 2 designs as arguments and returns a measure of "`how different" they are. This is used by a *diversity module* within GADO to weed out duplicates and designs that are very similar in order to encourage diversity in the population of potential designs. This helps the GA maintain a much thorough representation of the search space as well as allowing it to escape from local optima.

In the earlier runs, while the fitness function was under construction a simple diversity function based on the Euclidean distance between 2 strings was used. An example with 2 hypothetical strings of 4 *alleles* (numbers in the string) each illustrates the idea:

e.g. If the 2 individuals are A and B:

A:	11	3	0	1
B:	9	3	1	0

The 4-dimensional Euclidean distance is calculated as the square root of $\Sigma (A_i - B_i)^2$ where i =1 to 4.

$$= \sqrt{(11-9)^2 + (3-3)^2 + (0-1)^2 + (1-0)^2}$$
$$= \sqrt{6}$$

The problem with this simple distance function is that it is not sensitive to the objects in the design since distance between each allele is calculated on the same scale. After preliminary success with the GA, we set out to fix this. The idea was to use a weighted scheme corresponding to what each allele represented and how distinguishing a feature it was in the context of the entire circuit.

In the above example the sum might be:

Sum = $a|11-9| + b|3-3| + c|0-1| + d|1-0|$ where a, b, c and d are weights that reflect the importance of each allele in its contribution to the total distance between the given designs. This is the scheme we ended up using.

As discussed earlier the objects in the chromosome were sorted by their distance to the origin to eliminate aliasing. This is because in our representation, the exact layout may have multiple representations because the order in which the objects are placed in the string (Figure 1) is immaterial.

Our scheme still does not fix the problem of *apparent diversity*. There are many physical realizations of a given transistor schematic each of which may have its own unique sequence of alleles. The function needs to be smart enough to say that these designs are more similar than designs that may have smaller distances but represent a totally different schematic. This is because in this domain, small changes in the location, size or orientation of the objects can cause large changes in their logical equivalents. Such a function would have to be *schematic sensitive* by computing the distance between the graphs of the schematics. We are now investigating this and also a different representation scheme [7] that can make distance computations easier.

III. EXPERIMENTAL RESULTS

These are the results from our attempts to build an inverter. Figures 5,6 and 7 show MAGIC layout editor screenshots of the inverters designed by the GA for three different templates with different label configurations. The complete snapshots from the evolution of these circuits may be viewed at our website [19].

Once we were confident that a handcrafted solution existed for a given template, the GA was designated the task of finding it. Figures 8 and 9 illustrate the highlights in the evolution of the design of the inverter shown in Figure 5. The C code ran on a Linux box over many iterations. As the GA progressed it placed the designs that were "best so far" (individuals that were evaluated with the highest fitness) on our web page [19] so that we could monitor its progress. The screenshots you see here are some of the highlights in the evolution of the cell. The run lasted 3 hours (the next section discusses how this may be reduced) on a 1.2 GHz machine.

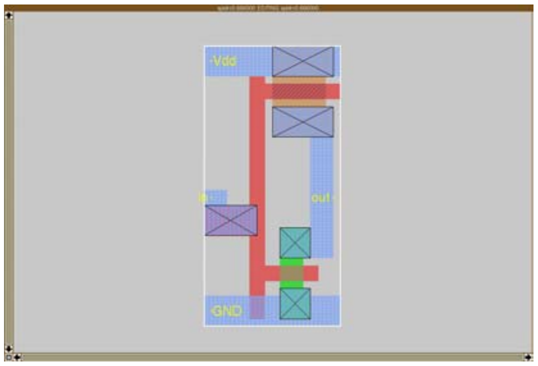

Figure 5: An Inverter Cell

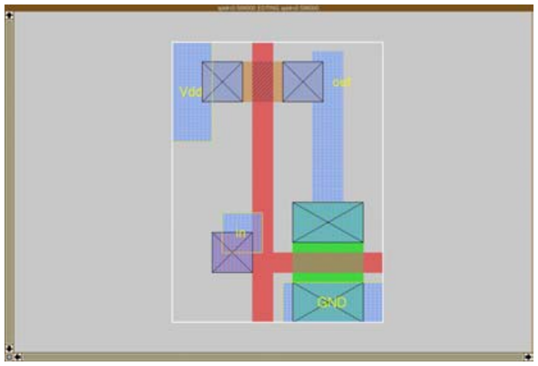

Figure 6: An Inverter cell with arbitrary label placement (1)

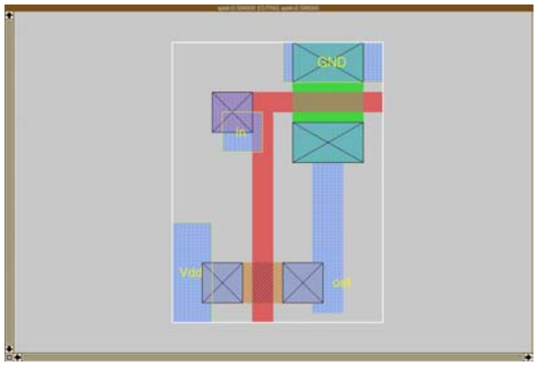

Figure 7: An Inverter cell with arbitrary label placement (2)

The first snapshot (Figure 8a) is that of a random placement of 6 (or as defined by user) objects from the building-block library (Figure 2).

Subsequent snapshots show the improvement of the circuit over time. The GA first weeds out overlapping transistors seen in Figure 8b and arrives at a simple design with a few objects and a lot of broken connections (Figure 8c). It then attempts to fix these connections by reducing the distance between nodes that could influence one another (Figures 9d-9g). On succeeding to fix all connections (Figure 9g),it attacks the final hurdle of coming up with a working circuit with zero design rule errors (Figure 8h). It then goes on to minimize the area (user defined criterion) occupied by this circuit and arrives at the final design shown in Figure 5.

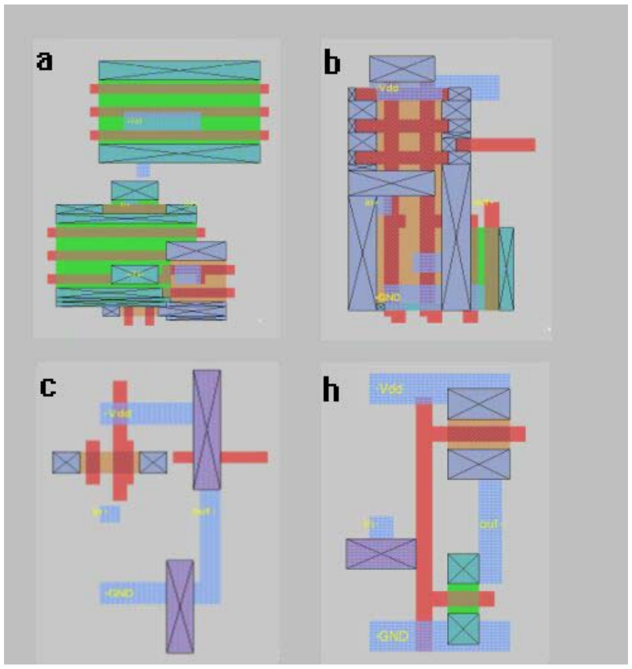

Figure 8: Highlights in the Evolution of the Cell (1)

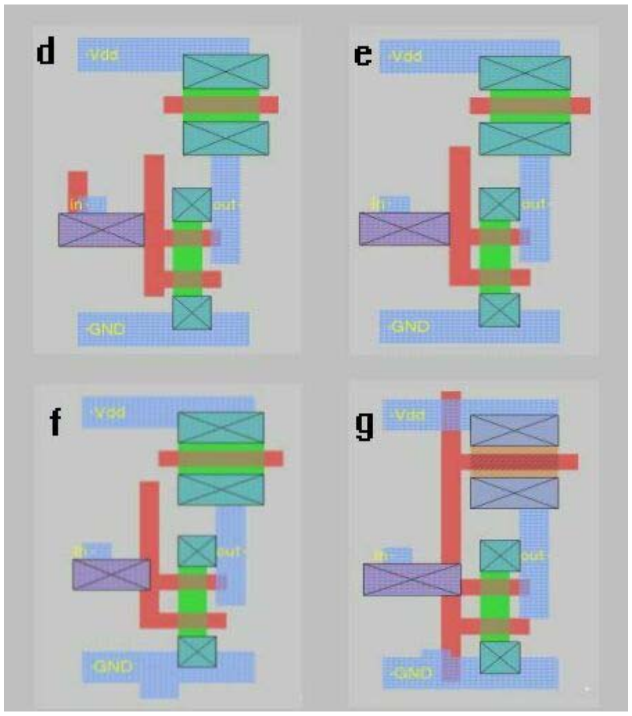

Figure 9: Highlights in the Evolution of the Cell (2)

IV. CONCLUSION

An evolutionary approach (using GADO, a genetic algorithm) for standard cell design automation was proposed.

A. LIMITATIONS

The GA can currently design cells with a small number of transistors. The search space becomes intractable as more and more objects are added. Plenty of optimizations can also be made within GADO. A better representation scheme is needed to tackle the combinatorial challenges of cells such as a full adder. As may be seen at our website, all runs of the GA do not converge at the "best" designs shown here. Different random seeds that the GA started with would converge at different designs. This means that the GA does not always find the global optima.

B. ADVANTAGES

GADO explores the space of all possible configurations (of a set of building blocks) given only a behavioral description of the circuit. The search space includes all possible electrical connectivity and layout and was accomplished in 3 hours on a single processor. The same result may be obtained in a fraction of the time by adding multiple processors since genetic algorithms are easily implemented on parallel architectures or a network of workstations [5,6,12].
Figure 10 summarizes our architecture implemented on a single processor.

The advantages of evolutionary methods over traditional path following methods are:

1. A more thorough representation of the search space through a whole set of points that collaborate to find better regions in the design space.
2. Intrinsically parallel and hence able to conduct a number of searches in parallel.
3. Easy to parallelize on a network of workstations.

This design methodology is an example of design by evolution. In addition, the *design* and *optimization* of the inverter happen in parallel at both logical (schematic) and physical (layout) levels. A working inverter was designed as a proof of concept.
This approach has the flexibility of generating cells on-the-fly to address the ad-hoc constraints faced by the designer when she is considering a candidate design in a larger context such as choosing a cell. This was demonstrated by the design of inverters with arbitrary label placements. This methodology would allow higher-level standard cell placement and routing tools to request cells with exact pin-orderings.

C. FUTURE WORK

Alternative representations are being investigated in an attempt to make the search space more tractable for more complex cells such as a full adder. We are also working on a better distance function that will avoid problems due to aliasing and apparent diversity. We also plan to test the algorithm on a network of workstations to reduce the run time of the algorithm.

D. ACKNOWLEDGEMENTS

This work was supported by Georgia State's Yamacraw initiative under the following grants: GCOMM2001.690130501-GIT and GCOMM2002.690000106-GIT.

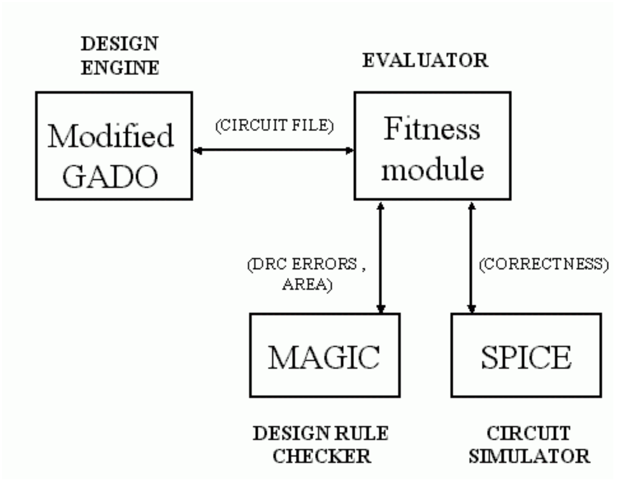

Figure 10: Architecture

V. REFERENCES

[1] M. Lefebvre, D. Marple and C. Sechen, "The Future of Custom Cell Generation in Physical Synthesis", *IEEE 34th Design Automation Conference*, July 1997, pp. 446-451.

[2] J. Burns and J. Feldman, "C5M – A Control Logic Layout Synthesis System for High-Performance Microprocessors", IEEE Trans. On CAD, 17(1), January 1998, pp. 14-23.

[3] B. Bishop, K. Rasheed and A. Bahuman, "VLSI Standard Cell Design Using Genetic Algorithms", *39th Annual ACM Southeast Conference*, March 2001,pp.

[4] K. Rasheed, "GADO – A Genetic Algorithm for Design Optimization", PhD. Thesis. http://www.cs.uga.edu/~khaled

[5] Goodman E. D., An Introduction to GALOPPS – the "Genetic Algorithm Optimized for Portability and Parallelism" System, CASE Cnter Technical Report #940401, Michigan State University, 1994, pp. 58.

[6] Gerald Carrier, Doyle Knight, Khaled Rasheed, and Xavier Montazel. "Multi-criteria Design Optimization of a Two dimensional Supersonic Inlet", 39th AIAA Aerospace Sciences Meeting and Exhibit, 2001.

[7] H. Murata, K. Fujiyoushi, S. Nakatake, and Y. Kajitani, Rectangle packing based module placement, *Proceedings IEEE International Conference on Computer-Aided Design*, pp. 472--479, 1995.

[8] D.G. Baltus, T. Varga and R.C. Armstrong , "Developing a concurrent methodology for standard-cell library generation", *IEEE 34th Design Automation Conference*, July 1997, pp. 333-336.

[9] MA Riepe and KA Sakallah, "Transistor Level Micro-Placement and Routing for Two-Dimensional Digital VLSI Cell Synthesis", *International Symposium on Physical Design*, April 12-14, 1999, pp. 74-81.

[10] K. Rasheed and B. Davison, "Effect of Global Parallelism on the Behavior of a Steady State Genetic Algorithm for Design Optimization", *Congress on Evolutionary Computation*, 1999, pp. 534--541.

[11] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, Chichester, England: Wiley 1990.

[12] P. Mazumder and E. M. Rudnick, *Genetic Algorithms for VLSI Design, Layout & Test Automation,* Prentice Hall, 1999, pp. 264-265.

[13] J. Rabaey, *Digital Integrated Circuits: A Design Perspective*, Prentice Hall, 1996.

[14] D.E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Reading, MA:Addison-Wesley, 1989.

[15] C. Edwards, *EDA Vendors Rethink Standard-Cell Libraries*, Electronics Times, June 2000.

[16] D. Pietromonaco, *Automating Cost-Effective Library Creation, Integrated System Design*, November 2000.

[17] http://www.research.compaq.com/wrl/projects/magic/

[18] http://bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE

[19] http://james.cs.uga.edu

[20] S.W. Mahfoud, A Comparison of Parallel and Sequential Niching Methods, *Proceedings of the International Conference on Genetic Algorithms*, pp. 136-143 (1995).

[21] M. J. Irwin and R. Y. Chen, Dynamic Circuit Synthesis Using the Owens Tool Set *11th Annual IEEE International ASIC Conference, September* 1998.