Proceedings of the 22nd ACM/IEEE Design Automation Conference, Las Vegas, NV, July 1985, pp. 702-708

Efficient Netlist Comparison Using Hierarchy and Randomization

J. D. Tygar[1] **Ron Ellickson**[2]

[1]Aiken Computation Lab., Harvard U., Cambridge, MA 02138.
[2]Valid Logic Systems, 2820 Orchard Pkwy., San Jose, CA 95134.

Abstract

Programs to compare the layout of ICs with their schematics have recently appeared. These programs have limited functionality and require large amounts of CPU time. We discuss the implementation of a fast $[O(n(\log n)^2)]$ logic comparison algorithm which uses hierarchy and randomization. This algorithm handles swappable components without perfomance degradation and is extremely robust in the presence of input errors. We include experimental data.

1 Introduction

Netlist comparison determines whether two circuits are identical, and identifies the differences if they are not. It is most commonly used for comparing netlists extracted from schematics with the netlists extracted from the corresponding IC layouts.

Historically, comparison of layouts and logic was done manually. Large plots were traced with colored pencils and laboriously compared with the schematics. This method was tedious and error-prone. Several years ago programs were developed that could extract a netlist from an IC layout, eliminating one source of error (*Alexander[78]*). Later, programs appeared that could compare the netlist extracted from the layout with a manually entered netlist. This reduced the possibility of comparison errors but depended on the quality of the human translation from schematic to netlist. Now designers can graphically enter schematics into workstations where the netlist will be automatically generated. At last the netlists from the layout and the schematic are available for comparison in machine readable form (*Scheffer-Apte[78], Barke[84]*).

As logic designs and IC layouts become larger, the comparison task becomes more difficult since comparison is CPU intensive and the netlist data requires increasing amounts of memory. Even when comparison is computationally feasible, conventional representations of netlist data make it difficult to isolate even very simple errors such as a power-ground short on large circuits since the short can occur in any subcircuit using power and ground nodes. Even worse, the presence of a single error can cause many spurious errors to be generated, propagating warning messages far out of proportion to the original error; much as a programming language compiler with bad error correction may report numerous warnings caused by a single error. The analogy is misleading because the situation with IC comparisons is much worse than with program compilations. Regardless of the verbosity of a compiler's error output, one can usually find the first error in a faulty program and fix it. Since there is no natural ordering of the netlists derived from a two-dimensional IC representation, there is no easy way to determine the true mistake from the error report.

With these faults of previous netlist comparators in mind, we set out to develop a new logic comparison program which attacked these problems on three fronts:

- *Hierarchy.* We allow the comparison to be done independently on any level of the hierarchy. This tremendously decreases execution time and isolates errors to a single subcircuit.
- *Randomization.* At each level of hierarchy, the netlists are compared by a new randomized algorithm that requires time $O(n(logn)^2)$ time (where n is the size of the netlist). In a typical design this gives two orders of magnitude savings over other comparison algorithms done on flat artwork. This algorithm works well whether the designer partially labels his nets or doesn't label any nets at all.
- *Enhanced Output Facilities.* If the circuits don't match we display the unmatched nets graphically; if the circuits do match we provide cross-referencing of additional information, such as parasitic capacitance, between the layout and the schematic.

2 Hierarchy

Our algorithm operates on two netlists. The netlists can be hierarchical or non-hierarchical. However hierarchy provides several advantages for netlist comparison.

Hierarchy allows users to compare parts of the design without having the schematic and layout completed for

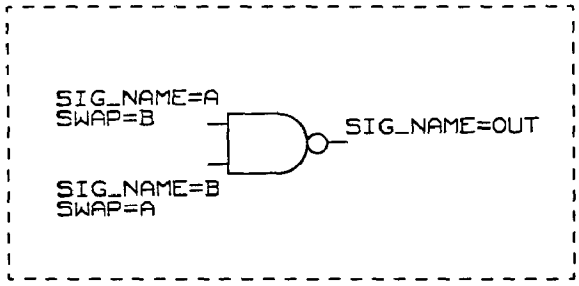

Figure 1: Swapping Terminals

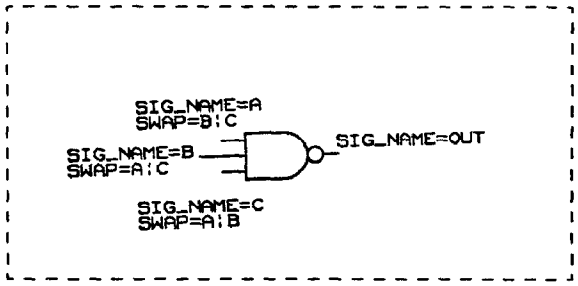

Figure 2: Swapping Three Terminals

the entire design. For example, two higher order cells can be compared without the lower order cells being fully specified. This is extremely useful for top-down chip composition; users can check the layout against the schematic step-by-step as they design the chip.

Hierarchy also helps to localize error detection and reporting to the cell being compared.

If a strict hierarchy with no overlap of included cells is used then additional benefits can be gained (*Scheffer[84]*). Strict hierarchical circuit descriptions not only promote good style, they also substantially simplify the task of netlist comparison. If each level of hierarchy contains information about no more than k subcircuits and nets and $F(n)$ represents the expected required time to compare two flat descriptions containing n subcircuits and nets, hierarchy reduces the time for comparison to $F(k)\log_k n$.

Hierarchical input presents a number of complications for the implementor. Because designers tend to construct subcircuits with inherent symmetries, sets of terminals attached to the subcircuit may be interchangeable. Instead of hard-coding specific interchangability (such as swapping DRAIN and SOURCE in a MOSFET) we generalized the algorithm to allow two different types of swapping.

Swapping Terminals

We allow the designer to indicate that terminals are swappable by attaching swap properties to them. For example, in figure 1, terminals A and B can be swapped. This implementation allows any number of terminals to be swapped. An example of a NAND gate with three swappable terminals can be seen in figure 2.

Swapping Sets of Terminals

Entire sets of terminals may also be electrically equivalent and therefore interchangeable with other sets. In figure 3 the set of terminals $< A, B, C >$ can be interchanged with $< D, E, F >$. This style of swapping is indicated by a property that is attached to the subcircuit. Furthermore, our implementation allows swappable information to be specified globally (for example, letting

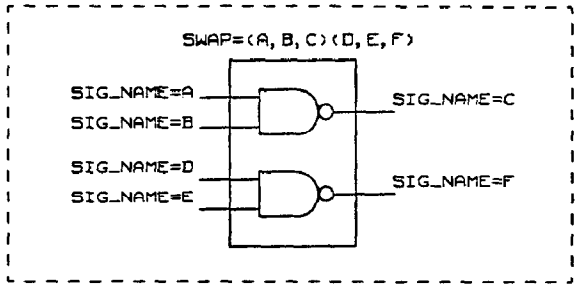

Figure 3: Swapping Sections

DRAIN and SOURCE swap on every MOSFET) or to be independently specified for each subcircuit. We have found that it is absolutely necessary for the compare program to allow swapping (without degradation of performance) if it is to be a viable tool.

3 Randomization and Comparison

Each netlist ℓ has an equivalent unique bipartite graph $G(\ell)$ defined by this procedure: Create a node in $G(\ell)$ for each cell or net in ℓ. If a cell is connected to a net by a terminal in ℓ, connect the corresponding nodes in $G(\ell)$ by an edge. If nets, cells, or terminals are named in ℓ, label the corresponding nodes or edges in $G(\ell)$ with the same name. (We denote the edges of $G(\ell)$ as $\mathbf{E}(G(\ell))$ and the nodes of $G(\ell)$ as $\mathbf{N}(G(\ell))$.)

We say that two netlists ℓ_1 and ℓ_2 *match* ($\ell_1 \sim \ell_2$) when there is an isomorphism between the graphs $G(\ell_1)$ and $G(\ell_2)$. If all the nets and cells in two netlists ℓ_1 and ℓ_2 are uniquely named, we can rapidly check whether $\ell_1 \sim \ell_2$. The unique names specify one-to-one functions; $f_{\text{edges}} : \mathbf{E}(G(\ell_1)) \rightarrow \mathbf{E}(G(\ell_2))$ and $f_{\text{nodes}} : \mathbf{N}(G(\ell_1)) \rightarrow \mathbf{N}(G(\ell_2))$. The netlists match exactly when f is an isomorphism. If the $\ell_1 \not\sim \ell_2$, the error may be easily isolated by finding which edges and nodes have to be added to $G(\ell_1)$ and $G(\ell_2)$ to make the graphs isomorphic.

Unfortunately, in real designs most nets are unnamed and several distinct instances of a cell may share the

same name since cells are typically named by their type. A netlist comparator must therefore construct the isomorphism function. Of course, the isomorphism function must still respect the labelling of the graphs — two labelled items can be correlated only if they have the same label. This is called the Partially Labelled Graph Isomorphism Problem.

If the $\ell_1 \not\sim \ell_2$, a good netlist comparator will construct a relation that is a "near-isomorphism", i.e., a relation between $\mathbf{E}(G(\ell_1))$ and $\mathbf{E}(G(\ell_2))$ and also between $\mathbf{N}(G(\ell_1))$ and $\mathbf{N}(G(\ell_2))$ that which to the greatest extent possible maps items in $G(\ell_1)$ into items in $G(\ell_2)$ that are isomorphic within some local subgraph. (We will make this notion more precise later in this paper.) As users, we measure of the quality of the error reporting of netlist comparators by the degree to which its relation correctly predicts what we, the users, intended to do. A more concrete measurement is given by the number of error messages the comparator produces on a given input; the fewer messages, the better the "near-isomorphism" relation.

It has been shown that if an algorithm P exists which solves the Partially Labelled Graph Isomorphism Problem in time $T(n)$, there exists an algorithm I which solves the Graph Isomorphism Problem (which is to check whether two graphs are isomorphic) in time $(T(n))^k$ for a fixed $k > 0$. Since the Graph Isomorphism Problem is widely believed to be intractable,[1] it would seem that development of an efficient algorithm for netlist comparison is infeasible (*Hoffman[82], Read-Corneil[77]*).

Despite these negative theoretical results, we can develop a fast isomorphism algorithm because circuits designed to be implemented in PC boards or IC's don't display the same generality that graphs do. We can exploit this fact by using randomized techniques.

Randomization is a relatively recent innovation in algorithms, which frequently makes problems which can not be efficiently solved using deterministic methods easily approachable (*Rabin[76]*). Given two netlists that satisfy our specifications, our randomized algorithm will never claim that ℓ_1 and ℓ_2 match when they don't; more important, the algorithm is guaranteed to find a match (if it exists) between netlists ℓ_1 and ℓ_2 in $O(n(\log n)^2)$ time with an arbitrarily high probability. For example, we could make the probability that the algorithm fails smaller than the probability that a meteor will suddenly fall from outer space and destroy the computer the algorithm is running on.

[1] Polynomial time solutions to the Graph Isomorphism Problem have been found for graphs of bounded valence (*Luks[80]*); however the exponent in the execution time of these algorithms grows very quickly with respect to the valence. Since we will be considering hierarchical circuits which may have cells with many connected terminals and nets which will have an unbounded number of connected terminals, these algorithms won't help us.

Restrictions on Netlists

We tailored the restrictions on our netlists to model the way that people actually design logic. The restrictions listed will not interfere with real practice. Let ℓ be a netlist, $n = \mid G(\ell) \mid$.

Restriction 1: There is an upper bound T_{cell} on the number of terminals connected to any cell in ℓ. (Physical characteristics prevent cells with unbounded number of terminals.)

Restriction 2: There is an upper bound T_{net} on the number of terminals connected to almost any net in ℓ; at most a fixed number B of nets exceed the T_{net} bound. (Stylistic considerations prevent this condition from being violated. If it is accidently violated, it can be easily discovered by error detection facilities.)

Let G be an arbitrary graph; $n_0 \in \mathbf{N}(G)$. The *i-environment* of n_0 ($\Delta(n_0, i)$) is the subgraph $S \subseteq G$ induced by n_0 and all the nodes in $\mathbf{N}(G)$ connected to n_0 by some path of at most i edges in G.

Recall that the graph G is said to have a *non-trivial automorphism* if there is an isomorphism function $f : \mathbf{N}(G) \to \mathbf{N}(G)$. The *unfixed nodes* of G (UNFIXED(G)) is the set

$$\{x \mid x \in \mathbf{N}(G),\ f(x) \neq x \text{ for some automorphism } f\}$$

Intuitively, the unfixed nodes are those nodes that can be switched by some automorphism.

Restriction 3: Partition UNFIXED($G(\ell)$) into connected sets $(S_1, S_2, \ldots)$. There are at most $v = O(\log n)$ automorphisms on each S_i. (If this condition wasn't statisfied, S_i would induce a complete graph (or a near-complete graph) of unbounded size. This is not a configuration which arises in real circuits.)

Restriction 4: If $m_0, n_0 \in \mathbf{N}(G(\ell))$, $m_0, n_0 \notin$ UNFIXED($G(\ell)$), then there is some $u = O(\log n)$ such that $\Delta(m_0, u) \not\cong \Delta(n_0, u)$. (Informally, this says that if two nodes cannot be interchanged, we can discover this fact by looking at the circuit within $O(\log n)$ edges of the two elements. Circuits failing this condition do occur sometimes; they are those circuits which have a repeated design normally handled by hierarchy. If for some reason they occur in a single (flat) level of hierarchy, the circuit can be matched in linear time by using the *zipper method* discussed below. For example, see figure 4. The nets at Y and Y-1 cannot be distinguished in fewer than $Y - 1$ elements. The zipper method distinguishes the two nets in linear time, however.)

Random Hashing

Our algorithm works by repeatedly hashing nodes in $G(\ell)$. Unfortunately, we have no way of knowing what the probability distribution of ℓ is. Therefore if we tried to fix a specific hash function h, we would have to assume worst-case behavior which might involve many collisions.

By choosing our hash function h from a set $H = \{h_1, h_2, \ldots\}$ we can get guaranteed good behavior with

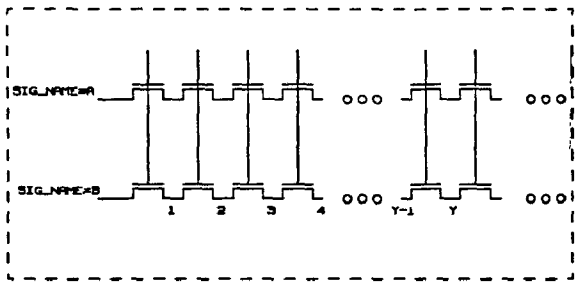

Figure 4: A Circuit Failing Restriction 4.

very high probability. What must the structure of the set H be? *Carter-Wegman[76]* answered that question by introducing the following definition and lemma:

Definition: Let H be a set of functions mapping set A into set B. H is *universal*$_2$ if for every $x \in A, y \in B$

$$|\{h \in H, h(x) = h(y)\}| \leq \frac{|H|}{|B|}$$

Lemma: Let A and B be sets. A universal$_2$ set H exists composed of hash functions from A to B. If $x \in A$ and $S \subset A$ and $h \in H$ is randomly chosen, then

$$\Pr(h(x) \in h(S)) \leq \frac{|S|}{|B|}$$

where Pr indicates expected probability.

We immediately have the following theorem:

Theorem: Given a ℓ satisfying our restrictions and fixed $i, c > 0$, there is a universal$_2$ H_i which constists of functions mapping $\Delta(n_0, i) \rightarrow [1, 2, \ldots, 2^c]$ (where $n_0 \in \mathrm{N}(G(\ell))$).

Proof: We proceed by induction. $\Delta(n_0, 0)$ is always the graph consisting of a single node which is distinguished only by name. According to the lemma, there is a universal$_2$ function mapping the name into $[1, 2, \ldots, 2^c]$. This is H_0.

Suppose $i \geq 1$, and we have constructed H_{i-1}. Using the unique numbering on the edges connected to n_0 implied the the unique numbering on the terminals of cells, consider the ordered pair

$$< m_1, \ldots, m_{\max(T_{\mathrm{cell}}, T_{\mathrm{net}})} >$$

of neighboring nodes of n_0. By **restrictions 1 and 2** these are all the neighbors of n_0 except in B cases. (We write $T = \max(T_{\mathrm{cell}}, T_{\mathrm{net}})$.) For those abnormal cases, use heuristics or guessing to uniquely identify the nodes and assign n_0 a random chosen integer $r_i \in [1, \ldots, 2^c]$ where $1 \leq i \leq B$ and i indexes the abnormal node. Otherwise, let $h \in H_{i-1}$ be randomly chosen, and compute

$$< h(m_1), h(m_2), h(m_3), \ldots, h(m_T) > .$$

Consider the universal$_2$ function X which maps T-tuples of values to $[1..2^c]$. X exists by the lemma; and the combination of X and H_{i-1} is a universal$_2$ map. Parameterize H_i by $X \times H_{i-1}$. □

Note that if $h \in H_i$ then h can be calculated in $O(ni)$ operations.[2]

Note further that if we were concerned with swappable pins, the above proof would have to be modified. If for example, terminals 1 and 2 were swappable for the cell corresponding to n_0, then instead of computing

$$< h(m_1), h(m_2), h(m_3), \ldots, h(m_T) >$$

we could compute

$$< h(m_1) \oplus h(m_2), h(m_3), \ldots, h(m_T) >$$

where $\oplus$ is, for example, addition modulo 2^c. (The discussion in *Reif-Tygar[84]* is helpful for understanding this.)

The Algorithm

Here is algorithm for determining whether netlists ℓ_1, ℓ_2 satisfying our restrictions match, and if they do, finding the isomorphism function f between $G(\ell_1)$ and $G(\ell_2)$:

1. Using **restrictions 3** and **4** fix $u = O(\log n)$ and $v = O(\log n)$. (Time: 0).
2. Compute $G(\ell_1)$ and $G(\ell_2)$. (Time: $O(n)$).
3. Randomly pick $h \in H_u$. Calculate h on $\mathrm{N}(G(\ell_1))$ and $\mathrm{N}(G(\ell_2))$. This randomly hashes all the nodes in the graphs according their u−environments. By the theorem and **restriction 4** all the nodes in the graph that are not unfixed have been assigned to unique values with probability $(1 - 2^{-c})^n$. If $x \in \mathrm{N}(G(\ell_1))$, $y \in \mathrm{N}(G(\ell_2))$, let $f(x) = f(y)$ if $h(x) = h(y)$. (Time: $O(n \log n)$).
4. With high probability, only unfixed nodes have not been matched uniquely. Partition the unfixed nodes into connected sets. (Time: $O(n)$).
5. By **restriction 4**, an isomorphism can be established between two specific nodes $x \in \mathrm{N}(G(\ell_1))$ and $y \in \mathrm{N}(G(\ell_2))$ if $h(x) = h(y)$. However, establishing a correlation between x and y may fix an isomorphism for elements in the connected sets that x and y lie in. Simultaneously fix an isomorphism for one pair of values in each of the connected sets, and assign a random value in $[1, \ldots, 2^c]$ as the hash value for that pair. Repeat steps 3 and 4. By **restriction 3** we will only have to repeat steps 3 and 4 $O(\log n)$ times at most. (Time: $O(n(\log n)^2)$).
6. Verify that f is an isomorphism function by walking through the graphs. (Time: $O(n)$).

[2] h can be quickly calculated in parallel. The theorem has several number-theoretic applications. A discussion of these issues will appear in a later paper.

Note that in the worst case the above procedure will take $O(n(\log n)^2)$ time. This procedure can fail only when the random hash function gives the same hash value to two nodes which can't be correlated in the given isomorphism function; when we execute step 3 we might misassign the isomorphism. This can only happen when an unlucky stream of random numbers are generated; there is no "fatal circuit" which causes the algorithm to run beyond its time bounds consistently. The probability of this procedure succeeding is $(1-2^{-c})^{nv} \geq (1-nv2^{-c}) = 1-O(n\log n)/2^c$ which can be made close to 1 by choosing large values of c. (In practice, of course, we handle the exceptional case by iterating until we make no progress, since it is much better to have the program exceed the time bounds than to give an incorrect answer.)

Zipper Method

Actual use of the algorithm revealed that it ran more quickly than the analysis above would suggest. Users tend to label subcircuits and nets liberally, avoid excessive automorphism, and use hierarchy effectively. Engineers seem to feel that circuits designed in these ways are easier to design, change, and understand.

These observations led us to a modification of the above algorithm. This modification also adds error-detection facilities. We dynamically maintain structures containing all the unmatched nodes. As we iteratively find corresponding nodes by the above algorithm, we remove the nodes from the list. When we find two matching nodes, we can speed the calculation of isomorphic points by exploiting structural properties. For example, if two cells are matched by the isomorphism function and the cells aren't connected to any swappable terminals, we can immediately match the nets corresponding to the given terminals. We can apply the same technique to the newly matched nets, and recursively match a large portion number of nodes. This is called the *zipper method* since the matching works like a zipper as it closes up the unknown parts of the isomorphism. This method copes very well with finding a "near-isomorphism" when the two netlists don't match. It matches a large number of cells and nets based on i-environments and then matches as much local information as the user designated. Since it cannot continue matching beyond where the isomorphism approximation breaks down, it tends to stop exactly where the user made the error. This robustness is one of the algorithm's most important features! Single errors often cause enormous problems to programs that work by zipper methods only.

Each time the algorithm matches two nodes from $\mathrm{N}(G(\ell_1))$ and $\mathrm{N}(G(\ell_2))$, it looks at the $H_0, H_1, \ldots, H_i$ values of all the neighboring nodes of the two picked nodes. If it finds a pair of neighboring nodes that have the same H_i value and no other node shares that value, it matches those two nodes. If two nodes have the same unique H_j value (which no other node shares) and different H_{j+1} values the algorithm matches the two nodes, guessing that the breakdown in the isomorphism function occurs in a node at distance $j+1$ from the newly matched nodes.

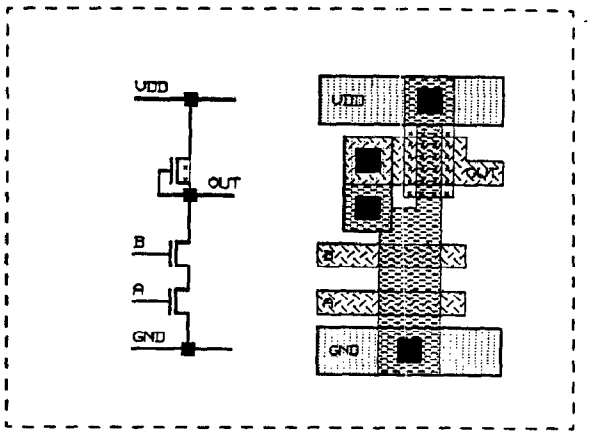

Figure 5: Circuit Example

4 Output

Netlist comparison is a second order program — it operates upon the output of other programs. Users enter their logic designs with a schematic capture editor and their layouts with a layout editor. Conceptually the input to the program is a set of graphics information. Our implementation of the compare program provides two methods of output: graphics output that is integrated with the graphics editor, providing a uniform interface for the user, and an ASCII listing which can be used by other programs.

Displaying Errors

In our implementation, errors are displayed by highlighting the non-matching portions of the two nets in the same editors that were used to create them. In figure 5 below the schematic matches the layout. In figure 6, net A in the schematic no longer matches net A in the layout and the error is highlighted.

Other Output

The isomorphism function constructed by our algorithm has many uses beyond merely demonstrating that two netlists match. Given the one-to-one correspondence between net in the two circuits, we can use the additional information supplied by component extraction. The extraction program gave us the netlist of the layout, but it also calculated additional information about the circuit, such as parasitic capacitance, resistance values, and

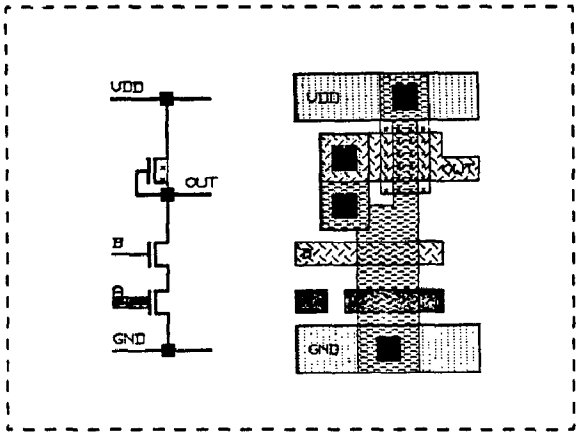

Figure 6: Highlighting Errors

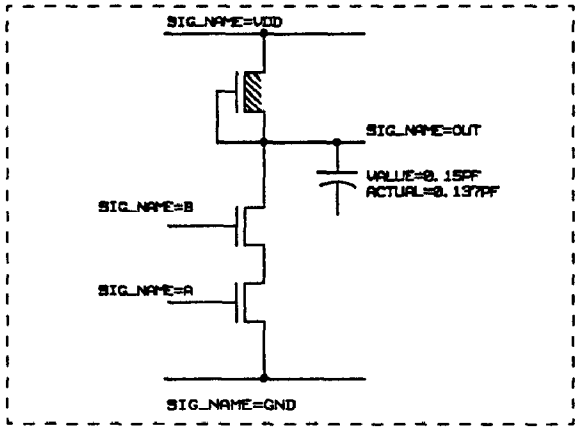

Figure 7: An Example of Backannotation

device sizes. The program backannotates the schematic with these values. This helps the layout engineer by documenting the actual capacitance load of the net. (See figure 7.)

More details on output format can be found in *Ellickson-Tygar[84]*.

5 Experimental Results

In addition to testing on "real world" cases, we wrote a program to generate sample netlists for comparison. This program generates a netlist with characteristics of typical MOS circuits ($\sim 1/4$ connections to VDD, $\sim 1/4$ connections to GND, rest random). It then scrambles the order of the parts, randomly swaps some swappable pins, introduces a known number of errors, and writes out another "near-isomorphic" netlist.

We first tested for running time vs. netlist size. We generated netlists with n transistors, $n/3$ nets, and only 2 labels (as if only VDD and GND are labelled.) Here are the results (times are for a 8MHz 68010 and 1 wait state):

Num. of transistors	50	100	200	400	600	800
Time/trans. (in ms)	51.0	56.5	59.3	79.4	69.3	243.3

Note that the running time behavior is close to linear until the hash table becomes saturated (the hash table has 800 locations). The hash table size could be arbitrarily extended. Since comparison in practice is hierarchical, this is not needed.

In the next test, we took almost unlabelled graphs (only 3 labels) and introduced errors by reconnecting one terminal of one device. In the best possible case, this should report only 2 unmatched nets. We tried 92 cases from size $n = 9$ to $n = 100$. The reported errors were as follows:

Num. of error messages	2	3	4	5	6	7	8	9	10
Frequency	85	3	3	0	0	0	0	0	1

So the program is robust in the case of the most common problems — lack of labelling and errors. Another example showe that this is true even when multiple errors are introduced. We introduced multiple number of errors in a random 100 transistor net list with no labels:

Num. of introduced errors	0	1	2	3	4	5	6	7
Num. of error messages	0	2	4	5	5	8	9	10

so the algorithm succeeds in matching most nets despite a large number of errors, even in the complete absence of labels. This behavior makes the program extremely useful for IC design work.

Another interesting test is to measure execution time vs. number of labelled nets. This test, with 100 transistors and 40 nets, shows that the algorithm depends very little on labelled nodes

Num. of labelled nodes	0	1	2	5	10	20	40
Time (in secs.)	5.6	5.6	5.6	5.7	5.5	5.4	5.6

The program delivers in practice what the algorithm promises in principle.

6 Conclusion

The comparison algorithm discussed in this paper works efficiently with hierarchical and non-hierarchical designs without the restrictions imposed by previous comparison algorithms. In particular, it is fast, robust in the presence of errors, and insensitive to the number of labelled nets.

Considerable care has been taken to implement a straightforward user interface. The input for the comparison is generated by integrated schematic and IC layout editors. The results of the comparison can be displayed with the same editors.

This compare program has been shown to be a tool applicable to many different design styles. It has been field-tested by users on many different circuits ranging from NMOS and CMOS microprocessors to digital bipolar chips.

7 Acknowledgements

We are indebted to Lou Scheffer for his many insightful comments and for his assistance in obtaining the experimental data. Also, thanks to Juan Leon, Roger Scott, and Mike Turner for their helpful remarks. The first author was supported in part by a NSF graduate fellowship and NSF grant MCS-81-21431.

8 References

Alexander[78] D. Alexander. "A Technology Independent Design Rule Checker." *Proc. 3rd USA-Japan Comp. Conf.* (1978) pp. 412 – 416.

Barke[84] E. Barke. "A Network Comparison Algorithm For Layout Verification of Integrated Circuits." *IEEE Trans. on CAD* 3 (1984) pp. 135 – 141.

Carter-Wegman[76] J. Carter and M. Wegman. "Universal Classes of Hash Functions." *Proc. 17th IEEE Found. of Comp. Sci.* (1976) pp. 106 – 112.

Ellickson-Tygar[84] R. Ellickson and J. D. Tygar. "Hierarchical Logic Comparison." *Proc. MIDCON '84* (1984).

Hoffman[82] C. Hoffman. *Group-Theoretic Algorithms and Graph Isomorphism.* Springer-Verlag (1982).

Luks[80] E. Luks. "Isomorphism of Graphs of Bounded Valence Can Be Tested in Polynomial Time." *Proc. 21st IEEE Found. of Comp. Sci.* (1980) pp. 42 – 49.

Rabin[76] M. Rabin. "Probabilistic Algorithms." *Algorithms and Complexity.* (J. Traube, ed.) Academic Press (1976).

Read-Corneil[77] R. Read and D. Corneil. "The Graph Isomorphism Disease." *J. Graph Theory* 1 (1977) pp. 339 – 363.

Reif-Tygar[84] J. Reif and J. D. Tygar. *Efficient Parallel Random Number Generation.* To appear; available as Harvard U. Tech. Rep. TR-07-84.

Scheffer[84] L. Scheffer. *The Use of Strict Hierarchy for Verification of Integrated Circuits.* Ph. D. Thesis, Stanford U. (1984).

Scheffer-Apte[78] L. Scheffer and R. Apte. "LSI design verification using topology extraction." *Proc. 12th Asilomar Conf. Circuits Syst. and Comp.* (1978) pp. 149 – 153.