Fast Incremental Circuit Analysis Using Extracted Hierarchy*

Derek L. Beatty†
Randal E. Bryant
Carnegie Mellon University

Abstract

We present an algorithm for extracting a two-level subnetwork hierarchy from flat netlists and its application to incremental circuit analysis in the COSMOS compiled switch-level simulator. Incremental operation is achieved by using the file system as a large hash table that retains information over many executions of the incremental analyzer. The hierarchy extraction algorithm computes a hash signature for each subnetwork by coloring vertices in a manner similar to wirelist-comparison programs, then identifies duplicates using standard hash-table techniques. Its application decreases the network preprocessing time for COSMOS by nearly an order of magnitude.

1 Introduction

The COSMOS switch-level simulator is an order of magnitude faster than other switch-level simulators executing on general-purpose hardware, at the expense of high pre-processing costs [4]. Pre-processing a network in COSMOS involves symbolic analysis of the network to produce Boolean descriptions of subnetwork behavior and interconnection of subnetworks, conversion of the Boolean descriptions to C-language code, and the subsequent compilation of that code. The compilation phase dominates preprocessing time. It would be possible to generate code in assembler language instead of C to eliminate the expensive compilation step, but this would make COSMOS non-portable.

The chief reason for long preprocessing times is that our symbolic analyzer ANAMOS is applied to flat network descriptions. This is necessary because each *subnetwork*, (i.e., each connected component of the channel graph corresponding to the network) must be analyzed [6]. A subnetwork consists of a set of nodes that can share charge and the transistors connecting them. Within a subnetwork, circuit operation may be complex and difficult to characterize due to charge sharing and the bidirectional nature of MOSFET switches. Thus, each subnetwork must be analyzed as a whole. Between subnetworks, however, signals are unidirectional, making behavior easier to characterize, so subnetworks can be analyzed independently.

ANAMOS processes flat network descriptions so as not to restrict the class of circuits COSMOS can simulate any more than necessary. There is no guarantee that an arbitrary hierarchical description would obey subnetwork boundaries. Circuits extracted from layouts may be flat [8] or hierarchical, but the hierarchy of the extracted circuit will depend on layout details such as cell overlap [12]. Designs containing structures such as busses will almost certainly violate subnetwork boundaries.

If the leaves of a hierarchical description respected subnetwork boundaries, each leaf could be analyzed only once, reducing the size of the Boolean descriptions and the amount of C-language code to be compiled. To speed processing, we *extract*, from a flat network description, a two-level hierarchy meeting this restriction. The leaves of the synthesized hierarchy obey subnetwork boundaries, and the higher level describes only the interconnection of subnetworks. This gives most of the speed advantage of hierarchical descriptions, without artificial network restrictions. This can be seen from Table 1, which shows processing times measured in minutes of CPU time on a VAX 11/780. Preprocessing without extracting hierarchy ("Flat" in Table 1) required 23 minutes. Generating initialized data structures in assembler[1] instead of the C language reduces this somewhat, but adding extraction of hierarchy ("Structured") reduces it further to 2.9 minutes.

The remainder of this paper describes our use of extracted hierarchy in COSMOS, presents the hierarchy extraction algorithm (concentrating on the differences between hierarchy extraction and the graph isomorphism algorithm on which it is based), and discusses other possible applications.

2 Incremental Analysis

Suppose there were a way to compute a hash function over subnetworks such that isomorphic subnetworks had identical

*This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, Amendment 20, under Contract F33615-87-C-1499, and in part by the Semiconductor Research Corporation under Contract 88-DC-068.

†Supported by a National Science Foundation Graduate Fellowship.

[1] Assembler-level data structures are significantly more portable than assembler-level code.

	Data Structure Format	
	C Language	Assembler
Flat	23.0	19.0
Structured	6.9	2.9
Incremental		
Cold	6.9	2.9
Warm	6.3	2.3

Table 1: Minutes to process ALU

hash codes, and that comparing subnetworks were very easy. Suppose further the existence of a *persistent hash table* (i.e., a hash table capable of storing information for much longer than a single execution of a program). It would then be easy to construct an incremental analyzer: given a subnetwork, it would compute the hash function, and check whether the circuit were in the persistent hash table. Were the circuit not found, it would analyze the subnetwork, placing the circuit and the results of analysis in the hash table. If, however, the subnetwork were found in the hash table, the program could simply read the results from the table.

The incremental analyzer just sketched assumes the existence of a persistent hash table, a means of hashing subnetworks, and a fast means of comparing them. Fast hashing and comparison can be accomplished by carefully relaxing the requirement of correctness: we allow two isomorphic circuits to have different hash codes, as long as this occurs infrequently; similarly, two isomorphic circuits may occasionally not be recognized as such. However, comparing two non-isomorphic circuits must always reveal that they differ. If comparing isomorphic circuits reveals their isomorphism often enough, the extracted hierarchy will be satisfactory, though it may contain some redundant leaves.

The hashing and comparison requirements could easily be met without relaxing correctness if circuits were placed in some canonical form, but finding such a canonical form is difficult. It is much easier to relax the correctness condition by coloring vertices, in the manner of a popular graph-isomorphism algorithm, then sorting the vertices by color to produce a *quasi-canonical* form. (A quasi-canonical form differs from a canonical form in that two isomorphic entities in quasi-canonical form are usually, but not necessarily, the same.) From this form, hash codes can be computed easily, and comparison can be done in linear time.

Note that the coloring process creates only a quasi-canonical form, for isomorphic subnetworks are not guaranteed to be identical after coloring. Were the form truly canonical, a simple comparison of two networks would reliably determine whether they were isomorphic. Though our form is only quasi-canonical, we compare networks simply, as if we had a truly canonical form. (A preliminary version of ANAMOS compared files containing colored subnetworks using a standard operating system program, `cmp`.) Thus, isomorphic subnetworks are sometimes missed, so the extracted hierarchy contains some redundancy. In practice, isomorphic subnetworks are seldom missed when analyzing small circuits. We do not believe that missed isomorphic subnetworks are significant in larger circuits (though any good way to measure this would imply the existence of a better way to extract hierarchy).

To implement the persistent hash table, we form a file name from each hash code. Each unique subnetwork and the results of its analysis are stored in files of this name with different types (i.e., filename extensions), in the *hash table directory*, an ordinary directory reserved for these files. If two distinct subnetworks hash to the same file name, we simply use the next available name.

The implementation of a persistent hash table requires careful attention to the operating system interface, and has been a significant source of difficulty in COSMOS. In most cases the speed advantages of incremental operation outweigh these problems, so ANAMOS defaults to incremental analysis, but incremental analysis can be disabled if necessary.

Incremental analysis allows COSMOS to exploit design latency and sharing. Particularly during final design stages when a design is nearly frozen, only small portions of the circuit will change with each revision. If the circuit is processed by COSMOS each time, each analysis after the first one will analyze only the incremental portions of the circuit that actually were changed. Portions that were not changed will be retrieved from the persistent hash table.

Incremental analysis may be undesirable at times, for a variety of reasons concerning the persistent hash table. If a design is undergoing frequent and major revisions, incremental operation may give little advantage, and the hash table directory may waste disk space storing obsolete subnetworks. Operating systems may cope poorly with a large number of files in a single directory. Finally, if a design is proprietary and is being encrypted whenever not being actively used, distributing parts of it onto many files may not be prudent. Many of these problems could be solved if the persistent hash table were implemented using database techniques, as discussed later. Since it is not always desirable to use the persistent hash table in its current form, ANAMOS can also use a more conventional internal hash table; in this case it extracts hierarchy but does not operate incrementally.

The savings due to incremental analysis are evident in the statistics shown in Table 1; processing this benchmark a first time ("Cold") requires 2.9 minutes, and processing it a second time ("Warm") requires only 2.3 minutes. The savings on a more realisitic circuit are even more apparent: the statistics in Table 2 were collected on a 43,000-transistor bus controller. When first given this circuit, ANAMOS required 94.3 minutes[2] on a lightly loaded VAX 11/780. Extracting hierarchy took only 15.9 minutes (17%) of this time; analysis of subnetworks required 20.4 minutes (22%), and the remainder was spent on global operations such as parsing the circuit description, constructing an internal representation, and writing the description of subnetwork interconnections. Processing the results of ANAMOS required an additional 140 minutes, 103 minutes of which was spent processing and compiling files describing individual subnetworks. The total time required to produce a simulator was about four hours.

When given this circuit a second time, COSMOS saves the time listed under "Nonrecurring" in Table 2. ANAMOS avoids

[2]Elapsed time was 94.3 minutes; CPU time was 53.1 minutes.

	Recurring	Nonrecurring	Total
Analyze	75.3	19.1	94.3
Generate	36.7	103.3	139.7
Total	111.0	122.4	234.0

Table 2: Minutes to analyze circuit and generate simulator for bus controller

analysis of subnetworks, saving 20.4 minutes immediately. Extracting hierachy the second time requires 17.3 minutes, only 81 seconds longer, because of the need to compare subnetworks to those already in the persistent hash table. The description of subnetwork interconnections produced the second time must be processed, but the subnetwork descriptions themselves need not be reprocessed, saving 103 additional minutes. The total time saved during reprocessing by incremental operation is 122 minutes.

The savings due to extracting hierarchy can be estimated from the times in Table 2, together with the bus controller statistics shown in Table 3. Extracting hierarchy is fast, requiring less than 25 milliseconds per transistor. The number of subnetworks analyzed is reduced by a factor of nearly 15, and the number of transistors by a factor of 4. Assuming that the time to analyze and process each subnetwork is constant, or constant per transistor, respectively, the 17.3 minutes spent extracting hierarchy save between 1714 and 367 minutes (or between 29 and 6 hours), i.e., extracting hierarchy reduces processing time by a factor of between 7 and 1.6.

	Transistors	Subnetworks
Flat	42940	6366
Hierarchical	10532	429

Table 3: Number of transistors and subnetworks in bus controller benchmark circuit

Extracting hierarchy so as to analyze each subnetwork only once improves preprocessing times in COSMOS by reducing redundant analysis and eliminating many procedures and the need for their compilation. We have, as mentioned before, chosen to generate assembler code to avoid the lengthy compilation of the initialized data structures that describe the interconnection of subnetworks. Many machines have similar formats for describing initialized data structures in their assembler language, so this does not cause significant portability problems.[3] The two techniques, extracting hierarchy to remove the procedure-compilation bottleneck, and generating assembler to remove the data-structure-compilation bottleneck, fit together nicely.

[3]Generating assembler-coded data structures may actually improve portability: several C compilers cannot compile the large initialized data structures generated by COSMOS.

3 Hierarchy Extraction Algorithm

The hierarchy extraction algorithm is based on a popular graph-isomorphism heuristic; details of the graph representation, initial vertex labeling, and vertex relabeling are adequately covered in [7], [9], [14]. In the following discussion, a *subnetwork* corresponds to a connected component of the circuit's channel graph; it consists of nodes and transistors. For each subnetwork there is a corresponding *coloring graph*, which is a bipartite graph containing a transistor vertex for each transistor in the subnetwork, a node vertex for each node in the subnetwork, and also a node vertex for each input to the subnetwork. Its (undirected) edges link transistors with nodes to which they are connected. Each vertex is labeled with an integer *color*; it is *pending* if any other vertex has the same color, and *uniquely labeled* otherwise. Pending vertices adjacent to vertices that have just become uniquely labeled are *frontier vertices*. (Ebeling and Zajicek [7] refer to transistor vertices as device nodes, and node vertices as net nodes; we prefer to use "vertex" for graph-theoretic discussion, and to reserve "node" for its traditional, circuit analysis sense.)

Extraction of hierarchy begins by finding subnetwork boundaries. (Subnetworks are easily identified, implying immediately that we do not claim to solve the NP-complete *subgraph isomorphism* problem.) Each subnetwork is then converted into a graph suitable for coloring, and its vertices are colored in a manner similar to that used by many wirelist-comparison programs. Vertices are initially assigned colors based on vertex properties invariant under isomorphism, then recolored by combining the colors of neighboring vertices using a hashing function. After each vertex has been colored uniquely, the nodes and transistors of the subnetwork are sorted by color of their corresponding vertices, yielding the quasi-canonical form. A hash code is computed for the quasi-canonical subnetwork, and the hash table is checked; if the subnetwork does not appear in the table, it is entered in the hash table and recorded as a leaf of the hierarchy. If it does appear in the table, it need not be recorded again; the subnetwork can be replaced by a pointer to the one found in the hash table. Bookkeeping code keeps track of the mapping between the original network and the quasi-canonical subnetworks.

Figure 1 gives a high-level view of the algorithm, which repeatedly identifies a subnetwork, labels the vertices of the coloring graph with unique colors, sorts the subnetwork by color, computes a hash signature for the subnetwork, and then compares the subnetwork to those it has previously encountered, using standard hash table techniques. It makes use of a straightforward procedure, `graph`, to convert subnetworks to coloring graphs, and it labels vertices using procedure `label_uniquely` shown in Figure 2. That procedure in turn calls two procedures, `refine_labeling` and `try_heuristic_labeling`, to assign vertex colors. Details of identifying subnetworks have been suppressed.

Figure 3 shows the procedure `refine_labeling`, which simply recolors frontier vertices, or if none exist, recolors all pending vertices. It terminates when all vertices are labeled uniquely, or when it is unable to label any nodes uniquely for several passes.

The procedure `try_heuristic_labeling` is shown in

```
while unprocessed subnetworks exist do
   N := an unprocessed subnetwork;
   (V,E) := graph(N);
   label_uniquely(V,E);
   sort N
   by labels on corresponding elements of V;
   if N is not in hash table then
      enter N in hash table;
      record leaf definition;
   fi;
   record instance of leaf;
od;
```

Figure 1: Algorithm for extracting hierarchy

```
procedure label_uniquely(V,E):
   b := 0;
   establish initial labeling of V;
   refine_labeling(V,E);
   while elements of V are not labeled uniquely
   do
      b := b+1;
      if (b > h_limit) then return;
      else
         try_heuristic_labeling(V);
         if heuristic labeling failed then
            return;
         fi;
      fi;
      refine_labeling(V,E);
   od;
end label_uniquely;
```

Figure 2: Procedure for labeling vertices uniquely

```
procedure refine_labeling(V,E):
   while elements of V are not labeled uniquely
         and progress is being made
   do
      if frontier vertices exist then
         recolor frontier vertices;
      else
         recolor
         all vertices not uniquely labeled;
      fi;
   od;
end refine_labeling;
```

Figure 3: Procedure for refining vertex labeling

Figure 4. It handles cases that cannot be labeled uniquely based only on circuit properties. The heuristic of labeling nodes by the *order* of node names, which works well in practice, is based on the observation that a designer will often assign similar node names to similar nodes of different subnetworks. Note that in the common case that a subnetwork contains many automorphisms, the heuristic simply selects some particular automorphism. The heuristic breaks only the smallest set of non-uniquely-labeled vertices, so as to use the least non-topological information needed to make progress. It always succeeds in establishing a unique labeling for each vertex, unless the circuit contains two or more transistors with identical gate, source, and drain connections.

```
procedure heuristic_labeling(V):
    if all node vertices in V
    are labeled uniquely then
        return failure;
    fi;
    find the node vertex equivalence class
    of smallest cardinality;
    sort the class by node name;
    assign sequential labels
    to all vertices in the class;
end heuristic_labeling;
```

Figure 4: Heuristic for breaking unlabeled classes

Several details complicate the algorithm. First, it is not always possible to label coloring graph vertices uniquely. Second, our storage of coloring graph descriptions in files, described earlier, complicates the hash table routines. Third, nodes and transistors of the original network must be bound to their corresponding entities in the reordered subnetworks.

It is not always possible to label coloring graph vertices uniquely, because many subnetworks contain sets of nodes which cannot be distinguished by vertex coloring. (Coloring graphs are isolated from the context in which they are used when they are colored, so information that might distinguish nodes if the circuit were colored as a whole, is sometimes lost. This makes indistinguishable nodes occur more frequently.) For some circuits (i.e., automorphic ones), this is of no consequence, but for others it does matter. For example, the inputs of the NOR gate shown in Figure 5a cannot be distinguished, but this is insignificant, since $\overline{a \vee b} = \overline{b \vee a}$. On the other hand, pairs of inputs of the AND-NOR gate of Figure 5b cannot be distinguished, which is indeed significant, since, e.g., $\overline{(a \wedge c) \vee (b \wedge d)} \neq \overline{(b \wedge c) \vee (a \wedge d)}$. Fortunately, heuristic labeling helps detect isomorphism in most such cases.

The correspondence between nodes of the original network and nodes in the extracted hierarchy is recorded as the hierarchy is constructed. Each node in an extracted leaf is referred to by its position in the quasi-canonical ordering. When each unique subnetwork is found, the list of those of its nodes that must appear in a higher level of the hierarchy, e.g., its inputs and outputs, is identified; this is simply a list of numbers. As each subnetwork of the circuit is colored, the ordering of its nodes is found. To instantiate the corresponding leaf, all that need be retained is a pointer to the leaf, and the ordered list of

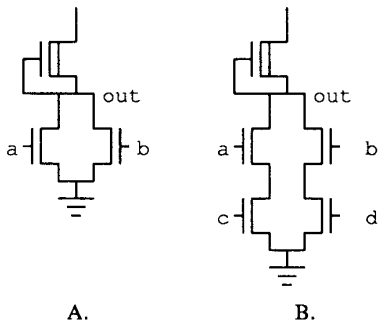

Figure 5: Circuits with indistinguishable nodes

those network nodes that are needed in the upper level of the hierarchy, i.e., those nodes whose numbers are recorded in the leaf.

It is important to note that this algorithm is not a fast solution to the difficult (NP-complete) subgraph isomorphism problem. The problems differ in three important ways. First, subnetworks are easy to identify from a network. Second, our algorithm sometimes fails to identify isomorphic coloring graphs, because of the tie-breaking stage. Finally, though our algorithm is very fast on the restricted class of graphs that correspond to actual circuits, it is not fast on general graphs.

A meaningful analysis of the algorithm is difficult. The space requirements are obviously negligible (i.e., linear). A worst case time bound of $\mathcal{O}(t^2 \log t)$ for each subnetwork containing t transistors is not difficult to derive, but it is not particularly meaningful; worst-case behavior is never observed in practice. Intuitively, unique colors propagate through a subnetwork by much the same paths as information would propagate through the actual circuit. Since circuits encountered in practice are designed to be fast, they can be colored quickly.

3.1 Related Work

Our vertex coloring is based on an algorithm used in many wirelist-compare programs. Many researches have presented such programs; Ebeling and Zajicek [7] give a very clear explanation. Kodandapani and McGrath [9] present many implementation details. Wong [14] uses graph-coloring and also checks hierarchical descriptions. It is quite natural to adapt this graph isomorphism algorithm because the colors assigned to vertices lead to a natural vertex ordering, and thus a quasi-canonical form.

Asymptotically faster wirelist comparison algorithms exist, but adapting them to extract hierarchy seems less natural. The principal difficulty is that in extracting hierarchy it is not necessary to prove isomorphism directly; instead, the goal is to create a canonical or quasi-canonical form from which it is very easy to prove isomorphism.

Kubo *et al.* [10] present a fast algorithm based on Hopcroft's $\mathcal{O}(t \log t)$ algorithm for refining a set partitioning [2]. However, it operates on a single graph constructed from the two networks being compared, whereas extracting hierarchy requires operating on one subnetwork at a time.

Tygar and Ellickson [13] present a careful analysis of a particularly attractive randomized algorithm for netlist comparison that runs in only $\mathcal{O}(t \log^2 t)$ time. However, their algorithm contains a randomized matching step, whereas we need a deterministic algorithm in order to put graphs in quasi-canonical form. Adapting this algorithm would seem to require careful re-seeding of the (pseudo)random-number generator.

To our knowledge, nobody has presented vertex coloring to find hierarchy from flat descriptions, or used a file system as a persistent hash table for incremental operation. Other incremental programs, such as Magic's circuit extractor [12], operate by examining the modification dates of files.

4 Other Applications

This algorithm obviously can be applied to compress flat network descriptions into smaller hierarchical form. It can also be applied to circuit analysis problems other than COSMOS, including other compiled simulators such as SLS [3], verifiers such as V [11], and to a general class of problems sharing the characteristics described below.

A modification of this algorithm could be used to adjust hierarchical network descriptions to ensure that their leaves respected subnetwork boundaries. This seems useful because, as alluded earlier, there is no best hierarchical description for a circuit. A hierarchy corresponding to a clean geometric decomposition of a chip layout differs from a hierarchy respecting subnetwork boundaries, and each of these may differ from a hierarchy corresponding to a clean RTL description of a design.

In general, this algorithm could be applied to many problems that operate on flat network descriptions, provided that they share three key characteristics. First, problem decomposistion must be relatively easy. In our application, identification of subnetworks is not only easy, it is a necessary step of symbolic analysis. Second, the subproblems represented by subgraphs must be independent: solution of one subproblem must be applicable to all isomorphic subgraphs. Finally, the subproblems must be non-trivial: the time spent coloring a coloring graph must be repaid by time saved by eliminating redundant work. In our application, each identification of a duplicate subnetwork eliminates not only its analysis, but the compilation of a C-language procedure, so the time is repaid handsomely.

An unexpected benefit to COSMOS users is the presence, after analysis, of the persistent hash table. It eases circuit debugging using the compiled simulator. Rather than store the entire network so that the user can examine small parts of it, the simulator can simply read only the portions requested. A user can ask the simulator for the status of any node, and the simulator will read the appropriate network fragment from the hash table, and print a portion of the circuit surrounding that node; it appears to store the original representation of the entire transistor circuit, just as source-level debuggers for optimized programs appear to manipulate the original program [15].

5 Conclusion

We have presented an algorithm for deriving a two-level hierarchical description of a circuit from a flat description, illustrated its efficiency with an example from COSMOS, and described the class of analysis problems for which the algorithm could prove useful. This work can be viewed as applying netlist comparision techniques to do more than compare two networks; in this it is similar to the extraction of capacitance from layout to annotate schematics in [13].

Adding extraction of hierarchy provided a significant performance improvement for our network analysis program ANAMOS, leading us to make incremental operation a standard feature of this program.

Using the file system as a persistent hash table to yield incremental operation has produced mixed results. Although incremental operation increases performance, and the persistent hash table eases circuit debugging, the persistent hash table is not a true database; consequently, many database issues are unresolved. Old files must be manually purged from the hash table directory, although accidental deletion of files from the table will merely result in their being recreated the next time the circuit is analyzed. Concurrent access by multiple users to the hash table directory could corrupt files in the unlikely event that one user is creating a description of a subnetwork while another is also accessing the same subnetwork. Programs incur the overhead of constantly converting between textual and internal representations. These problems could be addressed by using a more conventional database for the persistent hash table. Although our reliance on the operating system's compiler, assembler, and linker, which manipulate ordinary files, has prevented our taking such an approach, there is no fundamental reason prohibiting it.

We embedded hierarchy extraction in our network analysis program in order to perform incremental analysis. Equally interesting would be a standalone program able to create hierarchical descriptions from flat ones, or modify existing hierarchies, for it would be more widely applicable.

Acknowledgement

Carl Ebeling provided valuable suggestions for efficient implementation of vertex coloring.

References

[1] I. Ablasser and U. Jäger, "Circuit Recognition and Verification Based on Layout Information", *18th Design Automation Conf.*, ACM, 1981, pp. 684–689.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[3] Z. Barzilai, *et al.*, "SLS—a Fast Switch Level Simulator for Verification and Fault Coverage Analysis", *23rd Design Automation Conf.*, ACM, 1986, pp. 164–170.

[4] R. E. Bryant, *et al.*, "COSMOS: A COmpiled Simulator for MOS Circuits, *24th Design Automation Conf.*, ACM, 1987, pp. 9–16.

[5] R. E. Bryant, "Algorithmic Aspects of Symbolic Switch Network Analysis", *IEEE Trans. on Computer-Aided Design of Integrated Circuits*, CAD-6:4, July, 1987, pp. 618–633.

[6] R. E. Bryant, "Boolean Analysis of MOS Circuits", *IEEE Trans. on Computer-Aided Design of Integrated Circuits*, CAD-6:4, July, 1987, pp. 634–649.

[7] C. Ebeling and O. Zajicek. "Validating VLSI Circuit Layout by Wirelist Comparison", *Int'l Conf. Computer-Aided Design*, 1982, pp. 172–173.

[8] A. Gupta, "ACE: A Circuit Extractor", *20th Design Automation Conf.*, ACM, 1983, pp. 721–725.

[9] K. L. Kodandapani and E. J. McGrath, "A Wirelist Compare Program for Verifying VLSI Layouts", *IEEE Design and Test of Computers*, June 1986, pp. 46–51.

[10] N. Kubo *et al.*, "A Fast Algorithm for Testing Graph Isomorphism", *ISCAS 79*, 1979, pp. 641-644.

[11] D. S. Reeves and M. J. Irwin, "Fast Methods for Switch-Level Veritification of MOS Circuits", *IEEE Trans. Computer-Aided Design of Integrated Circuits*, CAD-6:5, 1987, pp. 766–779.

[12] W. Scott and J. K. Ousterhout, "Magic's Circuit Extractor", *22nd Design Automation Conf.*, ACM, 1985, pp. 286–292.

[13] J. D. Tygar and R. Ellickson, "Efficient Netlist Comparison Using Hierarchy and Randomization", *22nd Design Automation Conf.*, ACM, 1985, pp. 702–708.

[14] Y. Wong, "Hierarchical Circuit Verification", *22nd Design Automation Conf.*, ACM, 1985, pp. 695–701.

[15] P. T. Zellweger, *Interactive Source-Level Debugging of Optimized Programs*, Palo Alto: Xerox PARC, 1984.