VLSI DESIGN
1999, Vol. 10, No. 1, pp. 117–125
Reprints available directly from the publisher
Photocopying permitted by license only

Hierarchy Restructuring for Hierarchical LVS Comparison

WONJONG KIM* and HYUNCHUL SHIN

Dept. of Electronics Eng., Hanyang University, Ansan, Kyungki 425-791, S. Korea

(Received 7 September 1998; In final form 20 November 1998)

A new hierarchical layout *vs.* schematic (LVS) comparison system for layout verification has been developed. The schematic hierarchy is restructured to remove ambiguities for consistent hierarchical matching. Then the circuit hierarchy is reconstructed from the layout netlist by using a modified SubGemini algorithm recursively in bottom-up fashion. For efficiency, simple gates are found by using a fast rule-based pattern matching algorithm during preprocessing. Experimental results show that our hierarchical netlist comparison technique is effective and efficient in CPU time and in memory usage, especially when the circuit is large and hierarchically structured.

Keywords: LVS, layout, schematic, comparison, verification

1. INTRODUCTION

Recently, most designs are specified at the behavioral level. They are transformed to the register transfer level (RTL), to the gate level, and then finally to the layout description. The transformation procedure is automatic and/or interactive. The final design should be completely validated before manufacturing.

One major task of chip-level verification is consistency proof between the original schematic netlist and the one extracted from the layout. This verification problem can be modeled as a graph isomorphism problem. However, no efficient (polynomial time) algorithm has been found so far. Therefore, the isomorphism test can be executed by an exponential number of comparison operations, but this is not acceptable for circuits with thousands or more number of elements. Hence the main goal of any heuristic algorithm has been to reduce the complexity (the number of comparison operations) of the verification. A common method to achieve this objective is to partition the nodes of the two graphs into several groups with the same features. If the number of nodes in these partitions is small (possibly only one), the number of comparison operations decreases drastically and isomorphism can be tested in acceptable computing time for a large number of nodes [1].

*Corresponding author. Tel.: +82-345-400-4083, e-mail: wjkim@dslab.hanyang.ac.kr

1.1. Previous Methods

In simulation-based methods [2], only exhaustive simulation can guarantee the functional isomorphism between the two circuits. This is prohibitively costly in most cases and the simulation result differences cannot easily locate the erroneous parts of the circuit.

Many methods reported so far are based on the refinement algorithm [3, 4], in which the set of all nets and all components is partitioned into classes with homogeneous properties: the types of components and the number of adjacent components. The initial partition is successively refined by taking into account the properties of the neighbors. Elements (components or nets) in singleton classes of one circuit should be directly matchable with elements in the respective classes of the other circuit.

Most of the previous comparison tools perform flat-level comparisons [3, 4]. This was sufficient in the past as designs were less complex. These tools become inefficient and inadequate as the design size grows beyond a few million components. For most flat-level approaches, verification time grows as $O(N^m)$ where N is the number of components and $m \gg 1$. Memory requirements grow linearly with the size of the design. The problem with the flat-level approach is that it takes too long (40–50 hours) to verify about 3–4 million devices [6].

Hierarchical approaches have also been proposed for the circuit comparison problem [5–7]. Some methods require isomorphic hierarchies for the schematic and the layout netlists. This is not often the case, since in most cases the schematic hierarchy reflects the functional organization of the circuit, whereas the layout hierarchy is built based on its geometric structure. Spreitzer [5] approached the problem by modifying the hierarchies to make them isomorphic. Nonetheless the hierarchical methods cannot be used for all kinds of circuits. Pelz and Roettcher [6] proposed a hybrid approach of hierarchical pattern matching and refinement methods. But the complexity of the hierarchical pattern matching is in $O(n^n)$ in the worst case, where n is the flat target circuit size ($|components| + |nets|$).

To improve the handling of functional isomorphism, Spickelmier *et al.*, proposed the application of a rule-based expert system [8]. Pin permutations and functional equivalence conditions of subcircuits can be flexibly handled by the rule-based system. However, due to runtime and memory requirements, this method is limited to small-sized circuits.

1.2. The Features of Our Hierarchical LVS Comparison

In this paper, we propose a new hierarchical netlist comparison technique, which compares a hierarchical netlist and a flattened netlist, for layout verification based on refining and hierarchy restructuring. The features of our hierarchical LVS can be summarized as follows:

- It is a hierarchical comparison technique using a modified refinement algorithm. Hierarchical comparison methods are more efficient in CPU time and requires less memory than flattened comparison methods.
- The hierarchy is restructured to remove ambiguities for consistent hierarchical comparisons. When a subcircuit is used with some of its inputs connected together at a higher level in the hierarchy, it is difficult to find the subcircuit from the layout netlist. Finding subcircuits with power/ground connections and/or floating (unused) signals are also difficult. Especially, finding subcircuits which consist of multiple groups of disconnected devices requires almost exhaustive search. In these cases, we restructure the hierarchy by generating new modified versions of the subcircuits for consistent hierarchical comparisons.
- Simple gates are found by using a fast rule-based pattern matching algorithm. Most integrated circuits contain a large number of simple gates, such as inverter, NAND, and NOR gates. Finding simple gates using the refinement

algorithm is inefficient because they have only a small number of transistors. Therefore, we find those gates by using a fast rule-based pattern matching algorithm.

- Permutable pins are considered during comparisons.

2. HIERARCHICAL LVS COMPARISON

In this section, we describe the overall algorithm of our hierarchical LVS comparison method. Two netlists are used for the comparison: a hierarchical netlist from the original schematic design and a flattened netlist extracted from the layout. The overall algorithm of our hierarchical LVS comparison is shown in Algorithm 1.

```
ALGORITHM 1 Hierarchical LVS comparison
Read netlists;
Restructure the hierarchy;
Merge series tr's for both netlists;
Find simple gates in both netlists;
for(each subcircuit s from leaves to
  the root of the hierarchical netlist) {
   if (# used < Th_Used) {
      Expand s;
      continue;
   }
   Find candidates for s from the layout netlist
      and its corresponding key node k from s;
   if (# candidates ! = # used)
     Expand s; /* flatten it */
   else {
     for(each candidate c for s) {
        Verify image of s starting from c and k;
        if (Verification is successful)
           Replace matched part by s;
        else {
           Expand s; /* undo replacement and
              flatten it */
        break;
        }
     }
   }
}
```

The main part of our hierarchical LVS comparison system is based on the refinement algorithm. We find subcircuits from the layout netlist in bottom-up order. To find images of one subcircuit from the layout netlist, we use a modified version of SubGemini algorithm [9]. The algorithm is quite effective, but it assumes that the external nets of a subcircuit are not connected together at a higher level in the hierarchy. However, real designs of integrated circuits have many subcircuits with merged inputs which are connected together. Therefore, we find this type of subcircuits, and then restructure the hierarchy by generating new modified versions of the subcircuits, so that later hierarchical comparison becomes straight-forward. The hierarchy restructuring is described in detail in Section 3.

Now we describe major parts of Algorithm 1 in detail.

2.1. Merging of Series Transistors

Integrated circuits usually contain a great number of series transistors. Furthermore, their gate signals are permutable in many cases. To match the permutable signals and to reduce the complexity of the isomorphism checking, it is desirable to merge a set of series transistors into a new multi-gate device. Series transistors can be found by examining nets. A net connecting only two source/drain terminals of the same-type transistors (or series transistors) conforms a new multi-gate device. However, when the common net is also connected to another external terminal in a larger circuit, the transistors attached should not be merged into a multi-gate device. The merged series transistors are used later to find simple gates. Series transistors can be checked by visiting each net with degree of 2.

2.2. Finding Simple Gates

Most integrated circuits contain a large number of simple gates, such as inverter, NAND, and NOR gates. Since the simple gates are composed of only a small number of transistors, finding all of them by using the refinement algorithm is inefficient. Therefore, we have developed a fast rule-based pattern matching algorithm. We apply this algorithm for both netlists.

In case of CMOS circuits, inverters have two transistors of different types with a common drain signal and a common gate signal. The other drain of the *p*-transistor is connected to Vdd and that of the *n*-transistor is connected to Gnd. Note that the drain and the source of a transistor are interchangeable.

Series transistors can make NAND or NOR gates with corresponding parallel transistors. When a set of series transistors are *p*-type transistors with one of their sources connected to Vdd, we search for dual *n*-transistors, with the common drain signal, whose gates are connected to the gates of the series transistors. If all the gate-matched *n*-transistors are found, we replace the series *p*-transistors and all the gate-matched *n*-transistors by a NOR gate. When series transistors are of *n*-type, with one of their sources connected to Gnd, we search for a NAND gate, in a similar way.

This can be easily extended for other circuit types, such as domino or non-dual circuits.

2.3. Hierarchical Subcircuit Matching

The main part of our hierarchical LVS comparison consists of a recursive loop for finding subcircuits from the layout netlist. Subcircuits are processed in bottom-up order because a subcircuit can only be matched after all its child subcircuits are matched. All the sub-circuits in the hierarchical netlist are ordered by a breadth first search (BFS) algorithm. When there are several subcircuits in a hierarchy level, we process the most frequently used subcircuit first, so that the size of the layout netlist can be reduced as soon as possible. This procedure rebuilds the hierarchy from the layout netlist to match the given restructured schematic hierarchy.

We have used a modified SubGemini [9] algorithm for finding each subcircuit. It consists of two phases. In phase I, SubGemini identifies all possible matchable locations of the subcircuit in the layout netlist. It does this by applying a partitioning algorithm to both netlists. This procedure chooses a key node, *k*, in the subcircuit and identifies all possible nodes in the layout netlist which might match the key node. This set of nodes is called the candidate vector, *CV*. Phase I acts as a filter to reduce the number of instances that need to be checked. In phase II, SubGemini verifies whether there is an actual subcircuit at each location indicated by the candidate vector. It examines each node *c* in the candidate vector and attempts to find a mapping between nodes in the subcircuit graph and nodes in the layout graph, such that *k* matches *c*. This is done by initially postulating a match between *k* and *c*, and by labeling the two nodes with a unique label. Starting from these nodes, the algorithm simultaneously labels both the layout netlist and the subcircuit netlist such that labels of nodes match if and only if there is a valid mapping between the two graphs. If this procedure finds exactly matching labels in the layout netlist for all the nodes in the subcircuit, then a subcircuit has been found. Otherwise, the candidate node is a false candidate.

In this paper, we modified the SubGemini as follows:

Since the hierarchical comparison technique may be efficient when subcircuits are used many times in the higher level subcircuits, the algorithm optionally expands subcircuits which are used only a couple of times to the next higher level. The bound for the number of subcircuits used (Th_Used) is given by the user. When the value is set to 1, the algorithm attempts to find all the subcircuits hierarchically.

When the number of candidates (*nc*) for a subcircuit is different from that of subcircuits used (*nu*) in the schematic netlist after Phase I, some candidates may cause illegal matching. To prevent this problem, we check *nc* and *nu*. If they are different, we expand/flatten the subcircuit to the next higher level. Therefore, if a candidate matching fails, we expand the subcircuit in the schematic netlist and also expand previously matched parts of the subcircuit in the layout netlist. With this expansion of both netlists, we can verify the circuit hierarchically without losing consistency.

Figure 1 shows an example in which straightforward matching is not possible. The subcircuit CA shown in Figure 1(a) can be matched to the dotted block CA in Figure 1(b) which includes parts of subcircuits CB and CC. However, if *I*1 belongs to a subcircuit CB and NR2 belongs to another subcircuit CC, then CA in Figure 1(b) must not be matched to CA in Figure 1(a). Because the match is not consistent. *I*1 and NR2 can produce an illegal candidate for the subcircuit CA. In this case *nc* is larger than *nu*, which indicates an illegal matching. Therefore, we expand the subcircuit CA to the next level, if $nc \neq nu$.

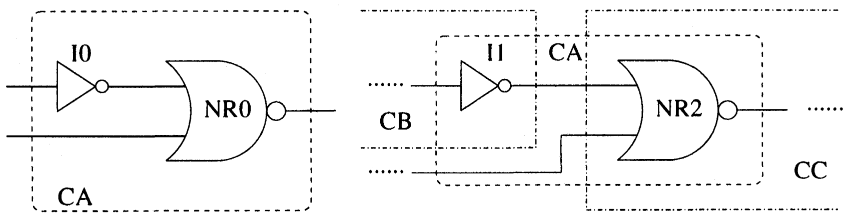
If there is an error for a subcircuit in the layout netlist then *nc* may be less than *nu* for subcircuits containing the errorneous part. Then those subcircuits may be expanded to the next higher level. After trying all the subcircuits to match, we can find the errorneous parts from the remaining netlist as described in [4].

Since these cases do not occur frequently, resolving this problem by exhaustively checking all the combinations of possible matchings is impractical. Therefore, we solve this problem by expanding the subcircuit and by performing matching at the next hierarchical level.

3. HIERARCHY RESTRUCTURING

During preprocessing, we restructure the hierarchy of the schematic netlist for consistent hierarchical matching. Subcircuits with power/ground inputs, merged inputs, or floating signals cannot be found directly by using the SubGemini algorithm. When a subcircuit consists of more than one connected group of devices, the refinement algorithm cannot be directly applied. Therefore, we generate modified versions of those subcircuits by exploiting external connections of them so that straightforward hierarchical matching can be used later. The hierarchy restructuring process is summarized in Algorithm 2.

```
ALGORITHM 2  Hierarchy restructuring
for(each subcircuit s from the root to leaves) {
  for (each device d in s) {
     Find power/ground signals;
     Find merged terminals;
     Find floating signals;
     if (any of them exists) {
       Try to find s' from the subcircuit list;
       if (s' doesn't exist) {
         Make a new subcircuit s' from s;
         Append s' to the subcircuit list;
       }
         Modify terminals of d;
```



(a) One subcircuit in the schematic (b) The subcircuit in the layout

FIGURE 1 Subcircuits which may cause a false matching.

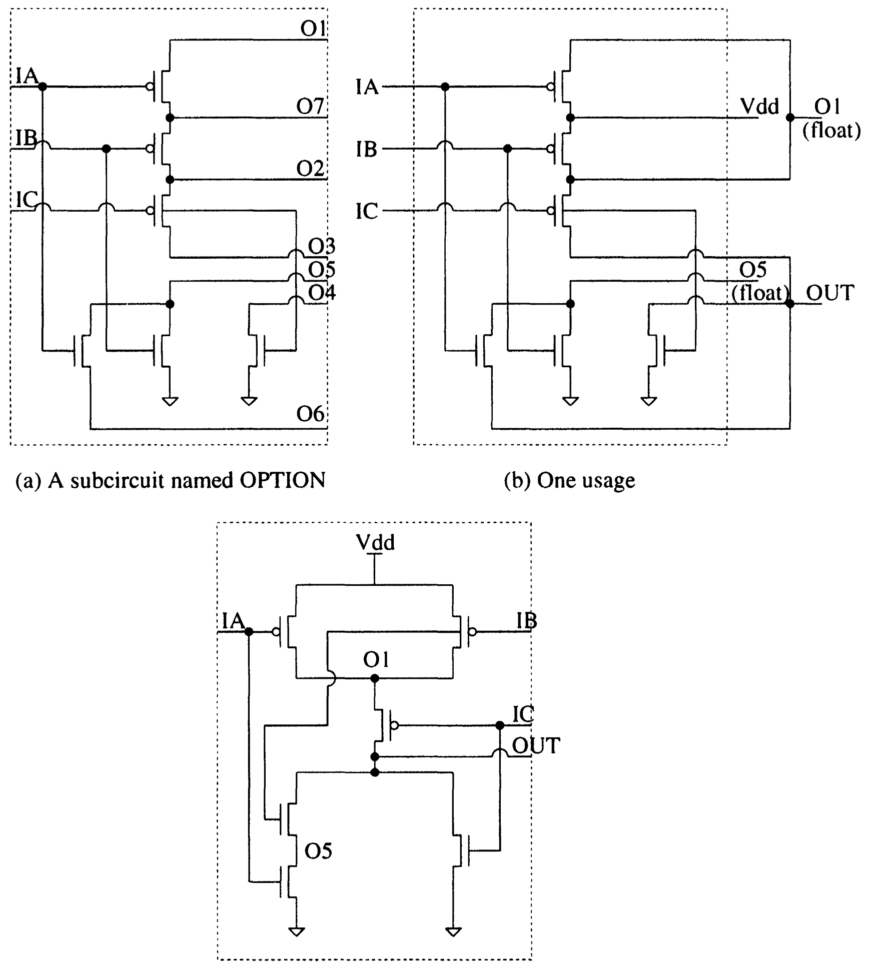
```
            }
        }
    }
for (each subcircuit $s$ from leaves to the root) {
    $n$ = number of disconnected groups in $s$;
    if ($n \geq 2$) {
        Split $s$;
        Expand $s$;
    }
}
```

Following subsections describe major parts of the algorithm in detail.

3.1. Subcircuits with Special Inputs

Subcircuits with special inputs can be found by exploiting external signals of each subcircuit in the hierarchical netlist. Some of these signals may propagate to their child subcircuits. Therefore, external signals are processed in top-down order.

Figure 2 shows an example of a subcircuit with special inputs. Figure 2(a) shows the original subcircuit OPTION. Figure 2(b) is an example usage of the subcircuit in a higher level subcircuit. Signals $O1$ and $O2$ are merged and the merged signal is floating. Signals $O3$, $O4$ and $O6$ are merged and is



(a) A subcircuit named OPTION

(b) One usage

(c) A modified version

FIGURE 2 An example of generating modified subcircuit.

connected to other devices. $O5$ is floating and $O7$ is connected to Vdd. Now the subcircuit will look like Figure 2(c) in the layout netlist. If we do not know that the signal $O5$ is floating, two transistors connected to $O5$ cannot be merged in the subcircuit because it is an external signal. However, in the layout netlist, two transistors connected to $O5$ can be merged because it has only two transistors, *i.e.*, $O5$ is never used outside of the subcircuit. We identify this case by exploiting the external connections in the hierarchical netlist and then generate a modified version of OPTION as in Figure 2(c) in the restructured schematic netlist. Then the circuit in Figure 2(c) can be matched with the layout netlist extracted as shown in Figure 2(b).

Floating signals of a subcircuit are propagated to child subcircuits. When a floating signal is connected to only one device in the subcircuit then it is propagated again as floating. However, if a floating signal is connected to more than one device, it is not floating any more. For example, if $F1$ and $F2$ are floating signals of the subcircuit 'C' in Figure 3, then $F1$ is propagated to the subcircuit 'A' as floating, while $F2$ is propagated to subcircuits A and B as normal external signals.

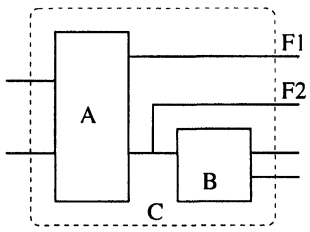


FIGURE 3 Propagation of floating signals.

3.2. Subcircuits with Multiple Groups

When a subcircuit consists of multiple disconnected groups of devices, it cannot be found by the Phase II of the SubGemini algorithm. We split this subcircuit into several connected subcircuits and it is expanded to the higher level. The higher level subcircuit also can consists of multiple disconnected groups. Therefore, we process each subcircuit in bottom-up order.

Figure 4 shows an example subcircuit. The original subcircuit E consists of subcircuits A, B, C and D. However subcircuit groups A-B and C-D are disconnected. In this case, we split E into $E1$ and $E2$ so that each of them contains A-B and C-D, respectively. Then we expand the subcircuit E which consists of $E1$ and $E2$. In this case, new subcircuits, $E1$ and $E2$ are generated, while E is removed, since E cannot be easily found.

4. EXPERIMENTAL RESULTS

We implemented our hierarchical LVS (HLVS) comparison algorithm on a Sun Ultra SPARC workstation running at 296 MHz by using the C programming language. We have tested our HLVS system on several industrial circuits and our own

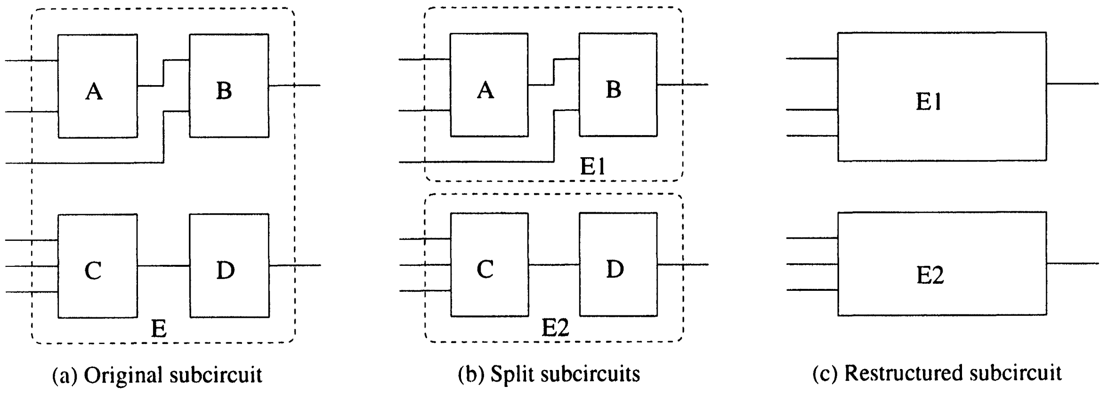


FIGURE 4 A subcircuit with multiple groups.

designs. For example, we verified circuits, such as a MCU with 27,091 transistors and a RAM with 1,328,692 transistors, in 6 seconds and 243 seconds, respectively. Table I shows the characteristics of the example circuits. Many different subcircuit types are used in circuits ex2 and ex3 while only a small number of subcircuit types are repeatedly used in ex4 through ex7.

Table II shows the experimental results of our hierarchical LVS and those of another well known algorithm, GeminiII (version 2.7, 1993). For GeminiII, the CPU times for flattening and comparison are shown in the parenthesis. Currently, the threshold value, Th_Used (explained in Subsection 2.3), of our HLVS system is 64 (by default), *i.e.*, a subcircuit is hierarchically processed if it is used more than 64 times in a circuit.

Note that the well-known commercial LVS tool, Dracula [10] (Rev. 4.3) took 395 seconds to verify ex2 (which was run by a layout expert in an industry), while our HLVS took only 27 seconds on a SPARC 20 workstation. The Dracula run time includes only LVS execution time, excluding database compilation and circuit extraction times.

Figure 5 shows CPU time variations for several large hierarchical circuits. It shows that the execution time of our algorithm increases almost linearly with the number of transistors, when the circuit is hierarchically designed.

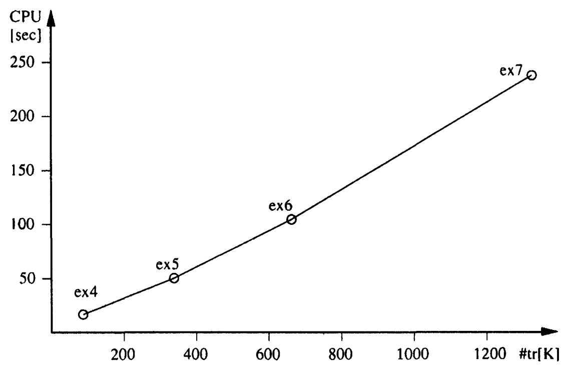


FIGURE 5 Linear characteristics of our algorithm.

These experimental results show that the proposed method is very effective and efficient.

5. CONCLUSIONS

We have developed a hierarchical LVS comparison technique which rebuilds the hierarchy, if possible, from the layout netlist by hierarchically applying the refinement algorithm. The hierarchical techni-

TABLE I The characteristics of the example circuits

circuit	# transistors	# subcircuit types	# total subcircuits used	# average subcircuits used
ex1	5,628	5	231	46
ex2	6,015	66	1,506	23
ex3	27,091	148	10,098	68
ex4	82,194	12	26,140	2,198
ex5	332,204	14	104,592	7,471
ex6	664,352	15	209,085	13,939
ex7	1,328,692	15	418,148	27,877

TABLE II Comparisons of experimental results

circuit	GeminiII		HLVS	
	CPU [sec]	Memory [MB]	CPU [sec]	Memory [MB]
ex1	1 (< 1+1)	2.6	< 1	2.3
ex2	< 1 (< 1+ < 1)	2.7	< 1	3.0
ex3	3 (1+2)	12	6	15
ex4	33 (6+27)	38	12	23
ex5	236 (62+174)	153	49	84
ex6	639 (239+400)	306	107	185
ex7	1,941 (1,032+909)	611	243	381
total	2,854	1,125.3	419	693.3

que is especially efficient when a subcircuit is repeatedly used. The hierarchy rebuilding may not be successful when there are ambiguous cases. Future works are necessary in this area. The parasitic effects, which are causing problems in UDSM designs, may be considered in future LVS [11].

Simple gates are found by using a fast rule-based pattern matching algorithm. Experimental results show that our hierarchical LVS approach is effective, especially when the circuit is large and hierarchically structured.

Acknowledgements

This work has been supported in part by Synopsys, Inc.

References

[1] Barke, E. (1984). "A Network Comparison Algorithm for Layout Verification of Integrated Circuits", *IEEE Trans. CAD*, **CAD-3**, 135–141.

[2] Preas, B. T., Lindsay, B. W. and Gwyn, C. W. (1976). "Automatic Circuits Analysis Based on Mask Information", In: *Proc. 13th Design Automation Conf.*, pp. 309–317.

[3] Abadir, M. S. and Ferguson, J. (1990). "An Improved Layout Verification Algorithm (LAVA)", In: *Proc. European Design Automation Conf.*, pp. 391–395.

[4] Ebeling, C. (1988). "GeminiII: A Second Generation Layout Validation Program", In: *Proc. Int. Conf. on CAD*, pp. 322–325.

[5] Spreitzer, M. (1990). "Comparing Structurally Different Views of a VLSI Design", In: *Proc. 27th Design Automation Conf.*, pp. 200–206.

[6] Pelz, G. and Roettcher, U. (1994). "Pattern Matching and Refinement Hybrid Approach to Circuit Comparison", *IEEE Trans. CAD*, **13**(2), 264–276.

[7] Batra, P. and Cooke, D. (1992). "Hcompare: A Hierarchical Netlist Comparison Program", In: *Proc. 29th Design Automation Conf.*, pp. 299–304.

[8] Spickelmier, R. L. and Newton, A. R. (1985). "Connectivity verification using a rule-based approach", In: *Proc. Int. Conf. on CAD*, pp. 190–192.

[9] Ohlrich, M., Ebeling, C., Ginting, E. and Sather, L. (1993). "SubGemini: Identifying Subcircuits using a fast Subgraph Isomorphism Algorithm", In: *Proc. 30th Design Automation Conf.*, pp. 31–37.

[10] Dracula Standalone Verification Reference Manual, Dec. 1994.

[11] Saleh, R., Overhauser, D. and Taylor, S. (1998). "Full-Chip Verification of UDSM Designs", In: *Proc. Int. Conf. on CAD*, pp. 453–460.

Authors' Biographies

Wonjong Kim received B.S. degree in Electronics Engineering from Chonnam National University, Korea, in 1989, and M.S. and Ph.D. degrees in Electronics Engineering from Hanyang University, Korea, in 1992 and 1999, respectively. He is a Post-Doc. researcher in the Digital Systems Lab. of Hanyang University. His research interests include computer-aided design of VLSI with special emphasis on high-level estimation, RTL-floorplanning, physical synthesis, and layout verification.

Hyunchul Shin received the B.S. degree in Electronics Engineering from Seoul National University, the M.S. degree in Electrical Engineering from the Korea Advanced Institute of Science and Technology in 1978 and 1980, respectively, and the Ph.D degree in Electrical Engineering and Computer Sciences from the University of California, Berkeley, in 1987. From 1980 to 1983, he was with the Department of Electronics Engineering, Kum-Oh Institute of Technology, Korea. In 1983, he received a Fulbright Scholarship. From 1987 to 1989, he was a Member of the Technical Staff at AT and T Bell Laboratories, Murray Hill, NJ. Since August 1989, he has been with the Department of Electronics Engineering, Hanyang University, Korea. His research interests include design and synthesis of integrated circuits and systems.