Hierarchical LVS Based on Hierarchy Rebuilding

Wonjong Kim and Hyunchul Shin
Dept. of Electronics Eng.
Hanyang University.
Ansan, Kyungki 425-791, S. Korea (R.O.K)
Tel: +82-345-400-5176
Fax: +82-345-406-9184
e-mail : wjkim@logos.hanyang.ac.kr
shin@heaven.hanyang.ac.kr

Abstract - A new hierarchical layout vs. schematic (LVS) verification system has been developed for layout verification. It compares a hierarchical schematic netlist and a flattened layout netlist. The schematic hierarchy is restructured for consistent hierarchical matching and then the same hierarchy is built from the layout netlist. For efficiency, simple gates are found by using a fast rule-based pattern matching algorithm. Each subcircuit is found from the layout by using a modified SubGemini algorithm in bottom-up fashion. Experimental results show that our hierarchical netlist comparison technique is effective and efficient in CPU time and in memory usage.

I. 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 finial 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 using 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].

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 can not easily locate the erroneous part of the network.

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 for nets. 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, 5]. 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 devices. For most flat-level approaches, verification time grows as $O(N^m)$ where N is the number of devices and m>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.

A. The features of our hierarchical LVS comparison

In this paper, we propose a new hierarchical netlist comparison technique 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 for consistent hierarchical comparisons. When a subcircuit has inputs merged 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, merged inputs, 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.
- The restructured hierarchy is rebuilt from the layout netlist by using a modified refinement algorithm.
- Commutable terminals 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.

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

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) {
  Find candidates for s from the layout netlist
    and its corresponding key node from s;
  if (#candidates != #used)
    Expand s;   /* flatten it */
  else {
    for (each candidate c for s) {
      Verify image of s starting from c and key;
      if (Verification is successful)
        Replace matched part by s;
      else {
        Expand s;        /* flatten it */
        break;
      }
    }
  }
}
```

modified version of SubGemini algorithm[9]. The algorithm is quite effective, but it assumes that the external terminals 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. 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.

A. Merging of series transistors

Integrated circuits usually contain a great number of series transistors. Furthermore, their gate signals are commutable in many cases. To match the commutable 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.

B. 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.

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 conform NAND or NOR gates. When one of series transistors is a p-type transistor with one of its drains connected to Vdd, we search for n-transistors with the common drain signal and their 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 its drains connected to Gnd, we search for a NAND gate, in a similar way.

This rule-based pattern matching is very fast and is not affected by merged inputs, power inputs, or floating inputs.

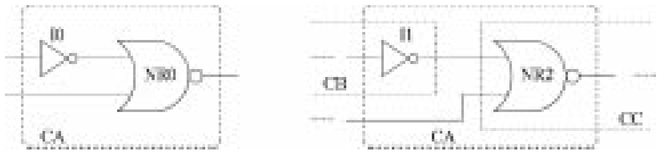
C. Hierarchical subcircuit matching

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 subcircuits 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.

Hierarchical netlist comparison is effective when a subcircuit is used many times. To capitalize this fact, we may optionally expand/flatten subcircuits which are used less than a certain number of times. The threshold value for subcircuit expansion can be given by the user. When the value is set to 1, the algorithm attempts to find all the subcircuits hierarchically.

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*, 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.

However, 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, we can verify the circuit hierarchically without losing consistency.



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

Fig. 1. Subcircuits which may cause a false matching

Fig. 1 shows an example in which straight-forward matching is not possible. The subcircuit CA shown in Fig. 1(a) can be matched to the dotted block CA in Fig. 1(b) which includes parts of subcircuits CB and CC. However, if I1 belongs to a subcircuit CB and NR2 belongs to another subcircuit CC, then CA in Fig. 1(b) must not be matched to CA in Fig. 1(a). Because the match is not consistent. I1 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 in the schematic netlist, if nc≠nu.

Resolving this problem by exhaustively checking all the combinations of possible matchings is impractical. Therefore, we solve this problem by flattening 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 can not 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 straight-forward hierarchical matching can be used later.

A. 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.

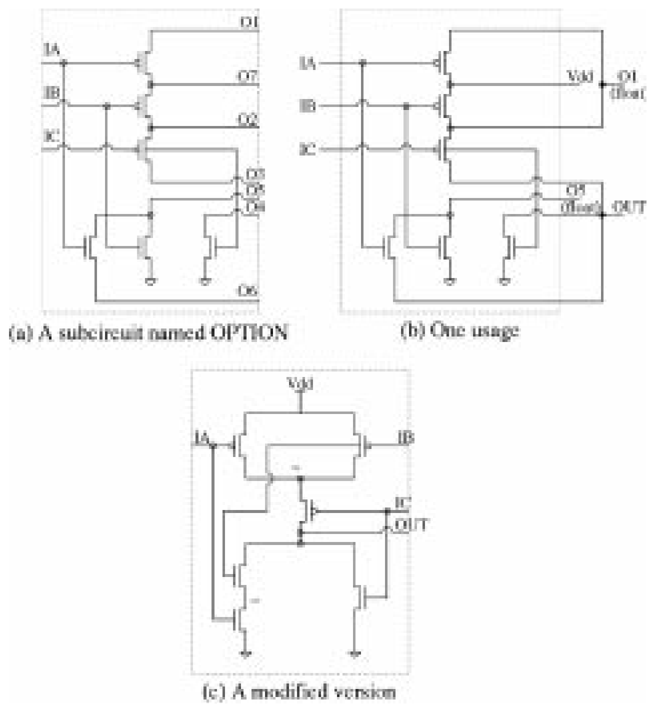


Fig. 2 An example of generating modified subcircuit

Fig. 2 shows an example of a subcircuit with special inputs. Fig. 2(a) shows the original subcircuit OPTION. Fig. 2(b) is an example usage of the subcircuit in higher level subcircuit. Signals O1 and O2 are merged and the merged signal is floating. Signals O3, O4 and O6 are merged and are connected to other devices. O5 is floating and O7 is connected to Vdd. Now the subcircuit will look like Fig. 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 Fig. 2(c) in the restructured schematic netlist. Then the circuit in Fig. 2(c) can be matched with the layout netlist extracted as shown in Fig. 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 Fig. 3, then F1 is propagated to the subcircuit 'A' as floating, while F2 is propagated to subcircuits A and B as normal external signals.

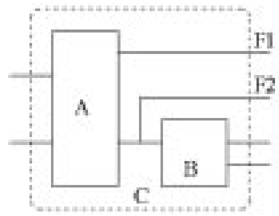


Fig. 3. Propagation of floating signals

B. 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.

4. Experimental results

We implemented our hierarchical LVS (HLVS) comparison algorithm on a Sun Ultra SPARC workstation by using the C programming language. We have tested our HLVS system on several industrial circuits and our own designs. For example, we verified circuits, such as a MCU with 27,091 transistors and a RAM with 666,366 transistors, in 10 seconds and 292 seconds, respectively. Table 1 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. Currently, the threshold value (explained in Section 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. The well-known commercial LVS tool, Dracula[10] (Rev. 4.3) took 395 second to verify ex2, while our HLVS took 27 second on a SPARC 20 workstation. This shows a significant speed up. Dracula was run by a layout expert in an industry. The Dracula run time includes only LVS execution time excluding database compilation and circuit extraction times. This shows that the proposed method is very effective and efficient.

5. Conclusions

We have developed a hierarchical LVS comparison technique which can rebuild the hierarchy from the layout netlist by hierarchically applying the refinement algorithm. For efficient hierarchical comparison, the given hierarchy is restructured when needed. 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] E. Barke, "A network comparison algorithm for layout verification of integrated circuits," IEEE Trans. CAD, vol. CAD-3, pp. 135-141, 1984.

[2] B. T. Preas, B. W. Lindsay, and C. W. Gwyn, "Automatic circuits analysis based on mask information," in Proc. 13th Design Automation Conf., pp. 309-317, 1976.

[3] M. S. Abadir and J. Ferguson, "An improved layout verification algorithm (LAVA)," in Proc. European Design Automation Conf., pp. 391-395, 1990.

[4] C. Ebeling, "GeminiII: A second generation layout validation tool," in Proc. Int. Conf. on CAD, pp. 322-325, 1988.

[5] M. Spreitzer, "Comparing structurally different views of a vlsi design," in Proc. 27th Design Automation Conf., pp. 200-206, 1990.

[6] G. Pelz and U. Roettcher, "Pattern matching and refinement hybrid approach to circuit comparison," IEEE Trans. CAD, vol. 13, no. 2, pp. 264-276, 1994.

[7] P. Batra and D. Cooke, "Hcompare: A hierarchical netlist comparison program," in Proc. 29th Design Automation Conf., pp. 299-304, 1992.

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

[9] M. Ohlrich, C. Ebeling, E. Ginting, and L. Sather, "SubGemini: Identifying subcircuits using a fast subgraph isomorphism algorithm," in Proc. 30th Design Automation Conf., pp. 31-37, 1993.

[10] Dracula standalone verification reference manual, Dec. 1994.

Table 1. Experimental results

circuit	#tr's	GeminiII		Our HLVS	
		CPU [sec]	Memory [MB]	CPU [sec]	Memory [MB]
ex1	5,628	3 (<1+3)	2.6	1	2.3
ex2	27,091	7 (3+4)	12	10	15
ex3	82,194	50 (12+38)	38	31	29
ex4	666,366	1,084 (508+576)	306	292	226
total	781,279	1,144	358.6	334	272.3