A Logic-to-Logic Comparator for VLSI Layout Verification

PETER M. MAURER AND ALEXANDER D. SCHAPIRA

Abstract—A logic-to-logic comparator (LLC) is a tool that verifies VLSI layouts by comparing the logic diagrams for a circuit with structures extracted from the circuit’s layout. LLC can operate at either the gate or the transistor level, although the gate level is preferred. It is similar to other graph-isomorphism-based tools, but incorporates some important improvements. The most important of these is a path-comparison algorithm that uses dynamically calculated global information. In addition, LLC contains a gate-matching algorithm that can use certain types of symmetry to distinguish between gates that appear to be identical to less sophisticated algorithms. LLC has been used to verify several commercially available VLSI chips including the WE® 32100 Microprocessor.

Index Terms—layout, verification, VLSI, graph isomorphism, symmetric gates.

I. INTRODUCTION

ALTHOUGH there are now many excellent tools for automatically laying out VLSI circuits, some portions of a design may still need manual intervention to solve timing and packing problems. A major problem in the design of such circuits is the verification of the manually laid out portions of the design. Such designs are usually specified as a set of logic diagrams that are manually translated into physical structures. One way to verify that the design has been realized properly is to compare the logic diagrams to the circuit’s physical structures. The comparison can be done either at the transistor or at the logic level. In either case, a technology-dependent tool is needed to either translate logic diagrams into transistor networks, or vice versa. This paper describes a tool that can be used to compare two logic diagrams or transistor networks to determine if they are equivalent. Although this tool can be used at either the logic level or the transistor level, it is targeted primarily at the logic level for reasons that will be explained later.

The problem of layout verification by network comparison has been extensively studied, and many algorithms have been developed [1]–[7]. Many of these algorithms are explicitly based on graph-isomorphism techniques [8]–[10], while others appear to be based on other methods. With few exceptions, however, all of these algorithms can be understood by relating them to the following generalized graph-isomorphism algorithm.

\[ S_1 = \text{initial\_partition}(\text{physical}); \]
\[ S_2 = \text{initial\_partition}(\text{logic}); \]
\[ \text{continue} = \text{true}; \]
while continue begin
\[ T_1 = \text{refine}(S_1); \]
\[ T_2 = \text{refine}(S_2); \]
if \( T_1 = S_1 \) and \( T_2 = S_2 \) then continue = false
else begin
\[ S_1 = T_1; \]
\[ S_2 = T_2; \]
end
end
if match\((S_1, S_2)\) then print “circuit verified”
else print list of mismatching elements

The “initial\_partition” function breaks the two circuits (physical and logic) into equivalence classes of circuit elements. The “refine” function refines these partitions by breaking existing classes into collections of smaller equivalence classes. Associated with each equivalence class is the list of attributes that distinguish the elements of the class from other circuit elements. The “match” function uses the attribute list (also known as the signature) to match elements with identical attributes. To avoid a combinatorial explosion, most algorithms restrict the “match” function to singleton sets, although some algorithms use backtracking to match the elements of nonsingleton sets.

Few algorithms exhibit a clean separation between the “initial\_partition,” “refine,” and “match” functions. Indeed, one can characterize each of the algorithms in terms of the power of the three functions, and the degree to which they are integrated.

Most of the algorithms use the local attributes of the gates and nets for the initial partition. For gates, some subset of type (MOSFET, NAND, etc.), fan-in, fan-out, and connection to known nets is used. For nets, node degree, and sometimes terminal type is used. Not all algorithms treat nets as explicit objects. In addition to local attributes, the LIVES system [7] uses distance distribu-
tion (also known as characteristic matrices) and distance from terminal nets for the initial partition.

In most cases, refinement, which is usually integrated with matching, proceeds by identifying an item with a unique signature in one circuit, locating its counterpart in the other circuit, binding the two elements, and relabeling the elements directly connected to the bound elements. In most cases it is possible to assign unique labels to the connecting elements. These labels can be used to enhance or replace the signatures of these elements. Once circuit elements have been bound, they are either removed from the circuits or exempted from further matching. Since relatively few circuit elements can be assigned unique signatures by the "initial-partition" function, the matching and refinement process is the primary method used to identify matching circuit elements.

It is important to note that this procedure causes refinement to be applied only to elements that are adjacent to elements with unique signatures. That is, the signature of an element changes only if it is adjacent to an element with a unique signature. This is in contrast to the refinement procedure used by more general graph-isomorphism algorithms such as that presented in [8]. The refinement procedure presented in [8] propagates information about unique signatures throughout the graph; however it is generally conceded that this procedure is too slow to be used with circuits of the size normally encountered in VLSI designs. The procedure described in the last paragraph will be called "local refinement" to distinguish it from the refinement procedure presented in [8]. Local refinement is the basis of the algorithms presented in [1]-[5], and [7]. In spite of its efficiency, it is easy to show that local refinement fails for many of the circuits that are encountered in practice. First, however, it is necessary to understand the role that symmetric gates play in the network matching process.

Symmetric gates are those whose inputs can be permuted in some way without changing the function of the gate. The most common examples are AND and OR gates. Layout verification algorithms differ in their ability to handle symmetric gates, and in the way that symmetry is used during the matching and refinement process. Most algorithms can handle totally symmetric gates (gates for which any permutation of the inputs is permissible) and some types of partially symmetric gates (gates for which any permutation of one or more subsets of inputs is permissible). A few algorithms are also capable of handling the nested form of symmetry exhibited by the AOI22 gate pictured in Fig. 1. It is permissible to exchange inputs A1 and A2 of this gate, and it is also permissible to exchange inputs B1 and B2. In addition it is permissible to exchange A1 and A2 as a group with B1 and B2. It is not permissible to exchange one of the A's with one of the B's. LLC can handle all of these types of symmetry, and in addition allows gate symmetries to be specified as arbitrary lists of permutations.

Symmetry is generally used in two different ways. It can be used to match symmetric gates whose inputs are specified in different orders, and once two symmetric gates have been matched, symmetry information can be used to assign (or avoid assigning) unique identifiers to the nets attached to a gate. LLC can use symmetry in both these ways, and can also use symmetry to uniquely identify gates that have the same signature. To see how this can be done, consider the three AOI22 gates illustrated in Fig. 2. In this example, LLC is attempting to match gate C with either gate A or gate B. Uppercase letters denote known bound nets. Since gates A and B are both bound to two known nets, X and Y, and are of the same type, they have the same signature. It is impossible to assign unique terminal types to the inputs of an AOI22, because the gate's symmetry allows any one input to be moved to any position on the gate. Nevertheless there are two distinct ways to attach a pair of inputs to an AOI22. Gates A and B of Fig. 1 illustrate these two types of connections. LLC's gate matching algorithm is capable of distinguishing between gates A and B and correctly matching A with C. This would enable LLC to bind the nets p and j, q and i, and k and r. The importance of being able to handle AOI22 gates will be discussed in greater detail below.

To return to the problem of local refinement, consider the logic diagram of Fig. 3. In this diagram, X and Y are totally symmetric gates. A, B, C, and D are known nets, and i, j, k, l, m, and n are unbound nets. This example is similar to several circuits we have encountered in practice. Abstract gate types are used to simplify the presentation. Although the two Y gates have unique signatures by virtue of being attached to different known nets, it is not possible to assign a unique signature to the inputs of these gates. In fact, it is not possible to assign a unique signature to any other element of this logic diagram using only local refinement. Nevertheless a careful examination of the diagram will show that it is not ambiguous. There is only one path from A to D that passes through an R gate, so that gate can be uniquely identified. This identification can be propagated to all other gates in the circuit, causing all elements to be uniquely identified.

Local refinement can fail even if global information is used during the initial partition. For example, the distance information used by the LIVES [7] system could be used to partition the diagram of Fig. 3 only if C and D are terminal nodes. If instead C and D were connected to a nonsymmetric gate of type Z as shown in Fig. 4, the R and Q gates could not be uniquely partitioned by the initial partition function, and local refinement would fail. This problem could be corrected by recalculating distance information during the refinement process, but this can also fail, as Fig. 5 shows. In this diagram the X gate is totally symmetric. Distance matrices cannot be used to partition this network because for every node at distance k from M or N there is an identical node at the same dis-
Fig. 2. Connection patterns for an AOI22 gate.

Fig. 3. A circuit for which local refinement fails.

Fig. 4. A circuit on which sophisticated local refinement algorithms fail.

Fig. 5. A circuit for which dynamically calculated distance matrices fail.

Fig. 6. Two nonisomorphic NMOS implementations of a NAND gate.

level or at the transistor level, or at some combination of both. LLC is no exception, but we believe that layout verification at the gate level imposes fewer restrictions on the circuit designer than does verification at the transistor level. To see why this is so, consider the two circuit diagrams illustrated in Fig. 6. Both of these circuits are NMOS implementations of a two-input NAND gate with inputs A and B, but the circuit diagrams are not isomorphic. If one applies a graph-isomorphism algorithm at the transistor level, it is necessary to artificially restrict the circuit designer to one circuit or the other. In order to get around this problem, our layout information is preprocessed by a tool called CTL which analyzes the serial and parallel connections of each circuit and outputs the equivalent network of logic gates. In addition to producing logic gates, CTL removes any redundant circuitry that was added to correct power problems, and performs a general sanity check on the transistors of the circuit. The input to CTL comes from a general-purpose artwork analysis tool [11]. A number of different design rule checkers are used to check circuit and transistor sanity before CTL and LLC are used.

Our motivation for developing LLC was twofold. First, our initial experimentation with pure local refinement algorithms uncovered many circuits of the type pictured in Figs. 3 and 4. It was clear that a more powerful matching and refinement algorithm was needed to handle these circuits. Second, it was necessary for us to be able to handle many different types of symmetry efficiently. Many of the circuits we needed to verify contained NAND and NOR gates with eight or more inputs. We needed to be able to handle totally symmetric gates with as many as 16 inputs without matching each gate against all permutations of the inputs. Furthermore our second most common gate after the inverter was a static D flip-flop. In our CMOS technology this gate is laid out as a two-input multiplexer followed by an inverter. One input to the multiplexer is the D input while the other is the feedback loop from the inverter. The multiplexer requires both the active high and active low versions of the clock. This circuit, which is pictured in Fig. 7, is a four-input circuit whose symmetry is identical to that of an AOI22 gate. Because this gate was so common, we needed the sophisticated symmetry-handling features of LLC to verify many of our circuits.

Section II of this paper gives an overview of the LLC algorithm. Section III describes the gate-matching algorithm. Section IV describes the net-matching algorithm,
the user. One option that can be selected is whether the first is preprocessing and initial partitioning, the second is refinement and matching, and the third is postprocessing and error analysis. The preprocessing step performs different functions depending on the options selected by the user. One option that can be selected is whether the tool is to use the logic designer's hierarchy when verifying the circuit. This option is useful for circuits such as memories that contain a large number of identical cells. When this option is selected, the tool verifies each subcircuit independently, and then verifies the overall connectivity. LLC allows circuits to be nested arbitrarily and allows gates and subcircuits to be intermixed in the same module. For most circuits the user will instruct LLC to ignore the logic designer's hierarchy. This will cause the tool to expand a hierarchical design into a flat circuit. (If it is desired to do verification at the transistor level, the user can supply a cell library at this point, and LLC will use the library to expand the gate network into a transistor network. To the best of our knowledge, this feature has never been used.) The reason the logic designer's hierarchy is usually ignored is because even though most logic designs and most layouts are constructed hierarchically, the hierarchies are seldom the same.

The second option that the user can select is to perform an initial binding between nets with identical names in the two circuits. For debugging purposes, the layout engineer can assign names to the various nets in the layout. If such names are used, it is common practice to use the same name used by the logic designer. As pointed out in [2] extensive use of such names makes the network comparison process trivial, but relying on names supplied by the user introduces a new source of error, and places a heavy burden on the layout engineer. In virtually all cases the user will instruct LLC to ignore the names assigned by the layout engineer, and create unique names for all nets in the layout.

The first step in the initial partitioning is to bind the input and output nodes of the two circuits. The logic simulation and layout tools used in our methodology require input and output nets to be declared in the same order. LLC uses this fact to perform an initial binding between all input and output nets. If the two circuits declare a different number of input or output nets, an error is reported and the declarations are ignored. The second step of the initial partitioning is to partition gates by type. Our methodology considers gates with different fan-in to be of different types regardless of the function performed. Connection to known nets is not used during the initial partitioning, but will be used during the refinement and matching step.

Each iteration of the matching and refinement step is broken into three substeps. The first of these is the gate-matching algorithm, which searches for gates with matching signatures and binds them along with the nets attached to them. The signature of a gate includes gate type, attachment to known nets, and, for symmetric gates, the pattern of known inputs to the gate. Although gates are logically partitioned by their signatures, no physical partitioning is done beyond gate type. This increases the execution time of the algorithm, but allows the binding of gates with similar but not identical signatures. If this type of binding were not allowed, an error near the inputs of a circuit could prevent the verification of correct logic in the center of the circuit. Since the algorithm can bind gates with differing signatures, bound gates are not removed from the circuits unless their signatures are identical and all connecting nets are bound.

Two nets can be bound only if neither of them has been previously bound. When nets are bound, the net of one circuit is renamed using the name of the net in the other circuit. Thus bound nets are those that have the same name in both circuits. In most cases when the gate-matching algorithm fails to bind any new nets, the next iteration will fail to bind any new gates since no gate signatures will have changed. Therefore, when the gate-matching algorithm fails to bind nets, the net-matching algorithm is invoked to locate and bind nets with unique signatures. Nets are partitioned based on in-degree, out-degree and connection type. In-degree is a count of the number of gates that use the signal as an input, while out-degree is a count of the number of gates that use the signal as an output. Connection type is determined by gate type and direction. No distinction is made between the inputs of a multiple-input gate, even for nonsymmetric gates.

If the net-matching algorithm fails to bind any nets, then the path-matching algorithm is invoked. This algorithm compares unique paths from known nets to known nets. When two matching paths are found, the nets in the paths are bound. This will normally allow the gate-matching algorithm to bind the gates in the path on the next iteration. Each of these algorithms will be explained in greater detail in the following sections.

The postprocessing and error analysis step is entered when all elements of both circuits have been bound, or when none of the matching algorithms can bind any new nets. The purpose of the postprocessing step is to bind gates that cannot be bound using the rules of graph isomorphism. The most important case is bidirectional trans-
mission gates that are specified in opposite directions in the two circuits. Another relatively common case is AND-OR-INVERT gates with duplicated inputs that are laid out as simpler OR-AND-INVERT gates. In all cases it is necessary that all nets connected to the gates be bound.

Transmission gates can cause problems because they are the only truly bidirectional gates used in our methodology. The circuit diagram for these gates is illustrated in Fig. 8. This gate allows the single input to be permuted with the single output. At this time LLC is unable to handle this type of gate in a satisfactory way. The postprocessing analysis of these gates is being used as an interim solution until a more satisfactory method can be developed. Another interim solution is to allow the preprocessing step to convert transmission gates into two-input zero-output symmetric gates, but this causes problems for the path-matching algorithm which cannot trace paths through the new gate. This problem is of fairly low priority because most of our circuit designers prefer to use the directional circuit pictured in Fig. 9 to implement tristate devices.

Error analysis is done by running the gate-matching algorithm with a special option that allows two gates to bind even if this would cause a rebinding of an already bound net. The net-binding portion of the algorithm is bypassed, and the pairs of bound gates are printed together to emphasize the differences. Finally, a list of the unbound gates from each circuit is printed. The overall flow of the LLC algorithm is illustrated in Fig. 10.

III. THE GATE-MATCHING ALGORITHM

The purpose of the gate-matching algorithm is to search the logic designer’s gates for the best possible match for each physical gate. For the purpose of this algorithm, gates are divided into three categories. The first category includes all totally symmetric gates, all gates that allow one or more subsets of inputs to be permuted arbitrarily, and all gates with nested symmetries. Gates in this category are known as commutative gates. The second category contains all symmetric gates that cannot be included in the first category. The symmetries of these gates must be described using an explicit list of permutations. The third category contains all nonsymmetric gates. There is a separate matching algorithm for each of the three categories.

A. Commutative Gates

Each commutative gate has a commutativity rule that describes the symmetries of the gate’s inputs (these are taken from a library rather than being hardcoded into the algorithm). For example, consider the AOI432 gate illustrated in Fig. 11. Fig. 11(a) contains an exploded logic diagram of the gate, while Fig. 11(b) illustrates the same gate as it would be described in our internal connectivity language. (In this language the format of a gate is (type): (name), ([input-list]), ([output-list]).) The commutativity rule for this gate is A, A, A, B, B, C, C, which allows inputs 1 through 4 to be permuted arbitrarily, but does not allow them to be permuted in any way with inputs 5 through 9. Similarly, it allows inputs 5 through 7 and inputs 8 and 9 to be permuted arbitrarily,
as long as inputs are not permuted between the two groups.

If an AND-OR-INVERT gate contains two or more commuting groups of the same size, the groups themselves may commute with one another. For example consider the AO122 gate illustrated in Fig. 12. The nested form of commutativity exhibited by this gate is described by the commutativity rule \( A(B, B), A(C, C) \). This rule allows the first two or the last two inputs to be interchanged and allows the first two inputs to be interchanged as a group with the last two. It does not, however, allow one of the ‘‘B’’ inputs to be interchanged with one of the ‘‘C’’ inputs. This form of commutativity is exhibited by several types of gates. LLC allows commutativity rules to be nested to any depth.

LLC uses the commutativity rule of a gate to define a collection of commutativity sets. Each distinct symbol in the rule defines one set, while each occurrence of the symbol defines one element of the set. Internally, LLC represents an element of a commutativity set as a pair of integers that denote the starting position and length of a contiguous set of inputs. For example, the rule \( A(B, B), A(C, C) \) defines three commutativity sets of two elements each, as illustrated in Fig. 13.

When two commutative gates are compared, a partial mapping is created between the inputs of the two gates. In order for the comparison to succeed, the partial mapping must be consistent with the gates’ commutativity rule, and it must be possible to extend the partial map to a total map in a unique way. These conditions are verified by using the partial map between the inputs to induce partial maps between commutativity sets. In the following discussion partial maps are written as sets of ordered pairs. If the set contains the pair \( (a, b) \) this implies that the partial map maps input \( a \) to input \( b \).

The procedure is best described by an example. Suppose the two gates of Fig. 14(a) are being compared. These gates have the commutativity rule \( A(B, B), A(C, C) \) which is parsed in Fig. 13. The partial map between the gates’ inputs is \( \{(1, 3), (3, 2)\} \). This map is illustrated in Fig. 14(a). (Recall that in order for two nets to have the same name in both circuits, the nets must be bound.) The consistency check begins with the commutativity set having the largest input count, which is set number 3. This set is derived from the portion of the commutativity rule that allows inputs 1 and 2 to be exchanged as a group with inputs 3 and 4. The induced map is \( \{1 \rightarrow 3, 2 \rightarrow 4\} \), where the numbers denote elements. This map is illustrated in Fig. 14(b). To be consistent, an induced map must be well defined (no two ordered pairs with the same first coordinate) and must be one-to-one (no two elements with the same second coordinate). If the mapping is not total, its order must be exactly one less than the order of the commutativity set. (The order of a map is the number of ordered pairs in the set describing the map.) This allows the partial map to be uniquely extended to a total map. If the induced map passes these three tests, it is used to rearrange the inputs of one of the
gates in preparation for processing the next commutativity set. Figure 14(c) shows the two gates after applying the induced map to the first gate. After applying the map, the new partial map between the inputs is \{(1, 2), (3, 3)\}. Since the two remaining commutativity sets have identical input counts, they may be processed in either order. Suppose set number 1 is processed next. The induced map is \{(1, 2)\}. It is vacuously well defined and one-to-one, and its order (1) is exactly one less than the order of the commutativity set (2). This map can be uniquely extended to the one-to-one map \{(1, 2), (2, 1)\}, which is illustrated in Fig. 14(d). Fig. 14(d) shows the gates after applying this map to the first gate. The processing of commutativity set 3 produces no further changes. Three net bindings can be generated from this pair of gates, Q-to-D, E-to-B, and T-to-X. These bindings are illustrated in Fig. 14(f).

The best match for a physical gate is selected from the set of all logic designer’s gates that pass the consistency checks. If there is a unique logic designer gate for which the partial map between inputs has maximum order, that gate is the best match. If there is more than one gate that produces a maximum-order partial map, then the output lists of the gates are compared to break the tie. If the tie cannot be broken by comparing output lists, no best match for the physical gate exists, and the gate remains unbound. The exception to this rule is gates that match completely. If two completely matching gates are found, they are immediately bound, regardless of whether there are other gates that completely match them. This permits LLC to bind gates that belong to one-gate parallel paths.

When two gates match completely, they are permanently removed from further processing, but if the match is partial, a potential match is created, and the logic-designer gate remains eligible for further matching with physical gates. When a previously matched gate is compared to a new physical gate, then the new match is ignored unless it is better than or equivalent to the existing match. A better match supersedes the previous match while an equivalent match causes both the existing match and the current match to be discarded. This procedure ensures that the best match for both gates has been found.

When a commutative gate has duplicated inputs, there may be more than one partial map between the inputs. If the commutativity algorithm chooses the wrong partial map, LLC may fail to match gates that should be matched. For example, consider the following two gates:

\[ \text{AOI33: } Y, (A, B, C, A, D, E), (X) \]
\[ \text{AOI33: } Z, (A, D, E, A, B, C), (X) \]

These gates are obviously identical, but if LLC chooses an initial partial map of \{(1, 1), (2, 5), (3, 6), (4, 4), (5, 2), (6, 3)\}, the consistency checks will fail and the gates will be considered unmatchable. To circumvent this problem, the input lists of all commutative gates are placed in canonical order before the matching algorithms are run. First the inputs of each commuting group are sorted into ascending sequence, then the sets are sorted by comparing the first pair of differing elements. This procedure is illustrated in Fig. 15.

**B. Other Symmetric Gates**

Symmetric gates that cannot be described by a commutativity rule may be described by a set of permissible input-list permutations. To be valid, this set of permutations must be a subgroup of all permutations on a set of \(n\) elements, where \(n\) is the number of inputs of the gate. However, LLC does not enforce this condition.

When a physical gate is compared to a logic-designer gate, the input-list permutations are applied one-by-one to the original physical gate. After each permutation is applied, the input lists of the gates are compared positionally. LLC selects the permutation that produces the most input-list matches. If more than one permutation produces the maximum number of matches, the two gates are considered unmatchable. The best match for the physical gate is selected from all matchable logic-designer gates. If two equally good matches are found, the output lists are compared to break ties. The matching process for noncommutative symmetric gates is illustrated in Fig. 16. Even though the symmetries of commutative gates could be explicitly specified as lists of permutations, it is not desirable to do so because a totally symmetric \(n\)-input gate has \(n!\) permissible input-list permutations. Since our technology permits us to build **AND** and **NOR** gates with as
many as 16 inputs, this method quickly becomes intractable.

C. Nonsymmetric Gates

For nonsymmetric gates, the input and output lists are compared positionally. The best match for a physical gate is the logic-designer gate that produces the most matches. As in the other categories, if two best matches are found, both matches are ignored.

IV. THE NET-MATCHING ALGORITHM

When the gate-matching algorithms fail to produce any net bindings, the net-matching algorithm is invoked. For each unbound net, LLC builds a structure containing information about the gates attached to the net, and whether the gates treat the net as an input or an output. Examples of net structures are given in Fig. 17(a). Fig. 17(b) shows the gate information used to build one of the net structures.

Unlike the gate-matching algorithm, the net-matching algorithm does not permit partial matches. In order for two nets to match, the structures describing them must be identical (except, of course, for name). Nets that are not unique in their circuit are excluded from the matching process. These nets will normally bind either by the path-matching algorithm or by later iterations of the gate- and net-matching algorithms. The combination of the net-matching algorithm and the gate-matching algorithm is more powerful than the gate-matching algorithm alone, as the logic diagram of Fig. 18 demonstrates. In this diagram, the gate-matching algorithm would be unable to distinguish between nets $x$ and $w$ or $y$ and $z$. The net-matching algorithm does so easily.

V. THE PATH-MATCHING ALGORITHM

When both the gate matching algorithm and the net-matching algorithm fail to produce any new net bindings, the path-matching algorithm is invoked. The path-matching algorithm constructs a set of paths through what remains of the two circuits, and compares paths between the circuits. Because in certain cases the construction of paths is subject to combinatorial explosion, the path-matching algorithm is invoked only when necessary. In most cases the path-matching algorithm will operate only on a small set of circuit elements that could not be bound by either of the other algorithms.

Paths are constructed by performing a depth-first search of the circuit starting from the outputs. For the purpose of this algorithm, an output is a bound net that is used only as an output by the gates remaining in the circuit. It is not necessarily a primary output of the circuit. Two mutually recursive subroutines are used to construct the paths. These are known as the gate processor and the net processor. The gate processor marks the current gate when it is invoked, and removes the mark when it terminates. When a previously marked gate is encountered, the gate processor returns without processing the gate. This protects the path constructor from going into an infinite loop, and implies that only simple paths will be constructed. The net processor makes a permanent copy of the current path when it encounters an input. For the purpose of this algorithm, an input is a bound net that is used only as an input by the gates remaining in the circuit. It is not necessarily a primary input. Because the gate processor does not record paths, each path must begin and end with a bound net. To protect against combinatorial explosion, the length of the maximum path is restricted to 17 elements. This number was obtained experimentally and seems to work well in practice. The construction of paths is illustrated in Fig. 19. Fig. 19(a) gives a logic diagram for which paths will be constructed, while Fig. 19(b) lists all the paths obtained for the diagram.

In order for two paths to match they must begin and end on the same nets, they must be of the same length, and the corresponding gate types must be identical. In addition, each path must be unique in what remains of its circuit. Once two paths have been matched, the corresponding nets are bound. The method for constructing paths guarantees that, except for the endpoints, none of the nets in a path will have been previously bound. Binding the
nets of a path will normally allow the gates to be bound on the next iteration of the gate-matching algorithm. See Figs. 3, 4, and 5 for examples of circuits that require the path-matching algorithm for verification.

VI. PERFORMANCE AND TERMINATION

When the path-matching algorithm fails to produce any new net bindings, the refinement and matching step is terminated. The refinement and matching step is guaranteed to terminate because there are a finite number of nets in the circuit. Since no net can be bound twice, the refinement and matching step will either bind all nets and terminate, or fail to bind any new nets on some iteration, and enter the error analysis routine.

Since each iteration except the last must bind at least one net, the maximum running time of the algorithm is \( N + 1 \) iterations, where \( N \) is the number of nets in the circuit. In most cases considerably more than one net will be bound on each iteration. In general the maximum number of iterations is usually \( D + 1 \), where \( D \) is the maximum distance between any net and an input or output node (there are exceptions either way). The gate-matching algorithm is \( O(G^2) \) where \( G \) is the number of gates in the circuit. A finer partition of the circuit could reduce the running time of the algorithm, but cannot change its fundamental quadratic character.

For nonsymmetric gates, the time required for each comparison is proportional to the number of inputs and outputs of the gate. For gates whose symmetry is described by a list of permissible permutations, the comparison time is proportional to the product of the number of inputs and the number of permutations. The comparison process for commutative gates is somewhat more complicated. The initial mapping between inputs is done using an \( O(i^2) \) comparison, where \( i \) is the number of inputs to the gate. For totally symmetric gates, this could be replaced by an \( O(i) \) algorithm since the inputs are sorted. The time required for verifying each commutativity set is \( O(i) \).

The net-and path-matching algorithms contribute little to the overall running time since, for most circuits, they are executed at most once. The net-matching algorithm requires \( O(c \log c) \) time to construct the net information, where \( c \) is the number of connections between nets and gates. (A simple sort algorithm is used.) At present the comparison algorithm is a simple \( O(n^2) \) comparison, where \( n \) is the number of nets. Future versions of LLC will probably use a more sophisticated algorithm.

The path-matching algorithm can require an exponential amount of time to create the list of paths, as Fig. 20 shows. However, this algorithm is usually invoked only after the gate- and net-matching algorithms have eliminated most of the elements from the circuit. In practice this algorithm seldom requires more than a small fraction of the total running time. Paths are matched using a simple \( O(p^3) \) algorithm, where \( p \) is the number of simple paths through the circuit.

These running times apply only to individual circuits.

If the logic designer's hierarchy is used during the comparison, the total running time is equal to the sum of the running times for each of the individual circuits.

Figs. 21 and 22 contain the results of a study that was performed to determine the effect of circuit size, circuit depth, symmetry, and errors on the running time of LLC. All tests were run on an AT&T 3B2/400 minicomputer, whose performance is roughly equal to 1.4 times that of a VAX 750. Execution times are reported in CPU seconds. The execution times reported in Fig. 21 were obtained by comparing circuits that contained from 1 to 64 chains of inverters. The lengths of the chains ranged from 1 to 64. The vertical dimension of this table shows the effect of increasing circuit size and circuit depth simultaneously, while the horizontal dimension shows the effect of increasing the circuit size while holding depth constant. The diagonal shows the effect of increasing circuit depth while holding circuit size constant. This study shows that increasing circuit depth has a much greater impact on execution time than increasing circuit size.

The execution times reported in Fig. 22 were obtained by comparing two 32-bit carry lookahead adders. The carry lookahead logic was implemented with three levels of 4-bit carry lookahead generators. Domino CMOS gates were used for all \&and, \&or, and \&and/or functions. The noninverting nature of these gates allowed the circuit to be built almost entirely of symmetric gates. The circuit contained a total of 212 gates, 180 of which were sym-
metric. Approximately 1200 transistors would be required to implement this circuit. The first comparison was done between identical circuits using an initial binding between nets with identical names in both circuits. The second comparison was the same but without the initial binding. Both of these comparisons were done with the symmetry handling features of LLC turned off. Symmetry handling was turned on for all other comparisons. The third comparison shows the effect of the symmetry handling features on the running time of the algorithm. Execution time increased by about 17 percent when symmetry handling was turned on. The fourth and fifth comparisons show the effect of a single error on the running time. With one incorrect gate type, the running time increased by approximately 15 percent, while with one incorrect input the running time increased by about 10 percent. For the incorrect gate type, LLC was able to bind all nets and gates except those that fanned out from the incorrect gate. For the incorrect input LLC was able to bind all circuit elements except the incorrect gate.

The sixth and seventh comparisons show the effect of multiple errors. For the sixth comparison a systematic error was introduced into all first-level carry lookahead generators. Each generated contained one gate with an incorrect input. This error increased the running time by about 50 percent. LLC was able to bind all circuit elements except the eight erroneous gates. For the seventh comparison, a massive error was introduced by switching the outputs of the propagate and generate functions. This had the effect of introducing wrong inputs into any gate that used these functions. This error increased the running time by 177 percent but this is hardly surprising since the error affected over half the gates in the circuit. In spite of this, LLC was able to correctly bind all nets in the circuit, and reported errors only in those gates that directly used the propagate and generate functions as inputs. Because all nets were bound correctly, the error was easy to identify from the error report.

Although the overall running time of LLC is well within what our users find acceptable, it is still larger than that reported for similar tools, in some cases much larger. One reason for this discrepancy is that the current version of LLC is a prototype that is partially implemented in UNIX® shell. In addition, most of the internal algorithms are implemented in rather naive ways. We believe that improvements in the LLC implementation will enable it to exhibit performance comparable to other similar tools.

Even if no performance gains were possible, we believe that LLC provides significant improvements over existing layout verification tools. The path-matching algorithm allows LLC to verify circuits containing subcircuits similar to that pictured in Fig. 4. Many of the circuits we have encountered in practice contain such subcircuits. In addition, the technique for matching gates allows totally symmetric gates to be compared efficiently regardless of width, and allows gates with nested symmetry to be uniquely identified even though they have the same nature. We have also found that the ability of the gate matching algorithm to bind partially matching gates helps isolate the errors in a circuit. In short, we believe that the improvements in functionality outweigh the decrease in performance.

VII. CONCLUSION

LLC has been used to verify the layouts of all chips of the WE 32100 product line. This line contains five chips including the WE 32100 CPU (a 32-bit microprocessor) and the WE 32106 MAU (an IEEE-compatible floating-point unit). Several versions of each chip have been designed. Because the layout of the earlier version of the CPU, the WE 32100 microprocessor, was verified using manually constructed high fault coverage tests, we were able to study the effect of introducing LLC into a virtually unrestricted design methodology. The only change required by LLC was a slight modification in the way that logic designers were required to specify certain inputs and output circuits. (The old method allowed the logic designer to use a logically correct but physically inaccurate circuit...) All of the designs were verified hierarchically using subcircuits of 100 to 1000 gates. In all but one case the subcircuits were verified without depending on net names supplied by the layout designer. The one exception was a subcircuit containing a highly interconnected network of 17 almost identical symmetric gates.

In summary, we have presented a new method for doing layout verification at the gate level. We have chosen the gate level rather than the transistor level because we believe it imposes fewer restrictions on the layout engineer. The method of matching and refinement is similar to other graph-isomorphism-based methods, with some important improvements. The first improvement is the path-matching algorithm, which allows dynamically calculated global information to be used during the refinement step. This allows us to verify many circuits that occur in practice that cannot be verified using only local information. The second improvement is a gate-matching algorithm that can use the connection pattern of gates with nested symmetry to distinguish between gates with identical signatures. The third improvement is the ability to bind partially matching gates, which permits errors to be isolated more effectively.

In practice, LLC has proved to be a highly effective tool that has enabled us to reduce costs and improve product quality. We believe that others will find these techniques equally beneficial.

ACKNOWLEDGMENT

The first author wishes to thank the University of South Florida Center for Microelectronics Design and Test for their support during the preparation of this manuscript.

REFERENCES


*Peter M. Maurer* received the B.A. degree in mathematics from St. Benedict's College, Atchison, KS, in 1969, and the M.S. and Ph.D. degrees in computer science from Iowa State University, Iowa City, in 1979 and 1982, respectively. He is an Assistant Professor of Computer Science at the University of South Florida. His current research interests include computer architecture, multiprocessing, VLSI design, verification, and testing; CAD tool integration; and wafer scale integrated circuits. Prior to joining the faculty of the University of South Florida, he worked for AT&T Bell Laboratories and AT&T Information Systems doing VLSI design verification and CAD tool development.

Dr. Maurer is a member of the Association for Computing Machinery, the American Mathematical Society, and the Mathematical Association of America.

*Alexander D. Schapira* received the B.S., M.S., and E.E. degrees in electrical engineering from the Columbia University School of Engineering & Applied Science in 1965, 1966, and 1972, respectively. He held the position of Preceptor in Electrical Engineering at Columbia University from 1967 through 1968.

In 1970, Mr. Schapira joined CompuScan, Inc., a manufacturer of optical character recognition (OCR) equipment. He developed algorithms, software, and hardware architectures for OCR systems, and he was granted a U.S. Patent for a new OCR system.

In 1981, Mr. Schapira joined AT&T Bell Laboratories as a Member of the Technical Staff in the CAD tools group of the Microprocessor Design Laboratory, Murray Hill, New Jersey. This organization subsequently became part of AT&T Information Systems as a result of the AT&T divestiture in 1985. He worked on CAD tools for VLSI layout verification, logic schematic generation and conversion, and logic verification and simulation.

After a short stay with the UNIX* Systems Development Department of AT&T Information Systems, at Summit, New Jersey, during which he worked on UNIX kernel enhancements, Mr. Schapira recently returned to AT&T Bell Laboratories in the IC Simulation Department, Murray Hill, New Jersey, and is looking forward in continuing work on IC design verification and simulation tools.

Mr. Schapira is a member of Eta Kappa Nu and the ACM.