Logic Verification in a Synthesis Environment[1] [2]

Wolfgang Kunz[3] *Dhiraj K. Pradhan* *Subodh M. Reddy*[4]

Max-Planck-Society
Fault-Tolerant Computing Group
University of Potsdam,
Potsdam, Germany

Dept. of Computer Science
Texas A&M University
College Station, TX, USA

Unisys Inc.
Mountain View, CA, USA

Abstract

A new methodology for formal logic verification of combinational circuits is presented. Specifically, a structural approach is used, based on indirect implications derived by recursive learning. It is shown that implications can be used to capture similarity between designs. This is extended to formulate a hybrid approach, this structural information is used to reduce the complexity of a subsequent functional method based on OBDDs. We demonstrate that OBDD-based verification can take great advantage of structural preprocessing in a synthesis environment where many small operations are performed that modify the circuit. The experimental results show that an effective combination can be achieved between memory efficient structural methods and powerful functional methods.

---

1 Research reported supported in part by NSF grant MIP 94-06946 and ONR grant #N00014-92-J-1366

2 A preliminary version of this paper was presented in DAC'95 and ICCAD'93

3 Portions of this research were conducted while this author was with Institut für Theoretische Elektrotechnik, University of Hannover, Germany

4 Portions of this research were conducted while this author visited with Max-Planck-Society, Fault-Tolerant Computing Group, University of Potsdam, Germany and are based on the thesis work of S. M. Reddy at Texas A&M University.

1 Introduction

Much progress has recently been seen in the automation of the design process for large integrated circuits. Tools for automatic synthesis play a crucial role in the progress of VLSI industry. With increasingly complex algorithms and software involved in VLSI design, the synthesis process can never be guaranteed to be completely free of errors. Any complex software can contain bugs and even a well-debugged tool, having worked correctly for many applications, may introduce unforeseen errors in future applications. Therefore, it is crucial to verify any circuit including those obtained by automatic synthesis tools against the initial specifications.

Design errors can also be introduced by needed interference of the human designer. Designers sometimes resort to manual modifications, or incorporate tailor-made software to fulfill the special design requirements. Often, changing a design at a later stage of the process is required, occurring if the specification has been modified slightly, or if a finished product must be updated to satisfy special requests. Such modifications are often referred to as engineering changes (ECs). Because ECs usually are conducted manually, this design process phase is both expensive and prone to errors. The designer must, therefore, have efficient tools enabling checking the functional correctness of designs after ECs.

The subject of this paper is the (formal) logic verification problem for combinational circuits; i.e., identifying whether two circuits are functionally equivalent. The presented approach is useful to verify circuits (or subfunctions of a circuit) after incremental design changes or ECs as they often occur in a synthesis environment. Further, an efficient verification method for such cases is also an important integral part if modifying and updating a given design shall become an automated process (*incremental* synthesis). Approaches to incremental synthesis and automatic design error corrections have been reported in [6, 12, 18].

Logic verification belongs to the most difficult problems in the field of computer-aided circuit design. Even verification of small designs can lead to enormous computation times, requiring large amounts of memory. The difficulty of the logic verification problem arises from having to explore the full functionality of both designs in order to formally prove their functional equivalence. Logic verification, therefore, usually is approached by *functional* methods; i.e. methods that try to completely capture the function of a circuit so that a comparison is possible. Much progress has been made with the introduction of Binary Decision Diagrams (BDDs) [2] and Ordered Binary Decision Diagrams (OBDDs) [8]. OBDDs permit a Boolean function to be represented in a compact and canonical form. If the circuits of comparison can be represented by OBDDs, the logic verification problem is solved; the circuits are functionally equivalent iff their OBDDs are isomorphic. Construction of OBDDs, however, is not always easily accomplished. The size of the OBDDs is highly sensitive to variable ordering, and for some circuits (like multipliers), the size of the OBDDs grows exponentially with the size of the circuit.

Structural techniques can be considered as an alternative to functional approaches. These techniques exploit the structural information of the given circuit implementations and their great advantage is that they operate on a structural gate-level netlist description of the circuit exploiting the structural properties of the design. Significant progress has been achieved in applying structural techniques to logic verification. As shown by in [5, 15], structural approaches to logic verification can perform extremely well, provided the circuits have some structural similarity. For circuits that are dissimilar these techniques can fail. BDD-based verification, on the other hand, is independent of the structural representation of the individual designs and does not rely on structural similarity. However, these methods too can fail because of exponential memory requirements. Therefore, in this paper, a methodology is proposed to effectively combine structural and functional methods for logic verification to mutually exploit the advantages of both paradigms. Our hybrid approach will be based on the structural method of [15] and a functional method using the OBDDs of [8].

2 The Incremental Nature of the Design Process

Because no general verification technique is known, which performs well for all classes of circuits, our interest is to investigate the use of special properties of practical verification problems and identify those that can be exploited in formulating effective procedures. Importantly, the suggested approach to logic verification exploits the general characteristics of a typical design process. A mathematical model describing the design process is presented in [1], schematically in Figure 1.

In [1], the design process is viewed as a sequence of "atomic operations" that constitute the different design steps. Such design steps either can be performed manually, or automatically by some synthesis tool. Each design step must be followed by an accurate and efficient verification step to achieve high design quality at low costs. It is important to note that the design process is of incremental nature as it consists of many small steps. This has important consequences for the verification steps because two successive intermediate designs, as they occur along the synthesis process, can be expected to have some degree of “similarity”. The fewer the atomic operations that are packed into a synthesis step the more similarity exists between the circuits. In particular, it can be expected that manual changes and ECs (they are particularly prone to errors) will let large portions of the circuit remain unaffected. Similarity refers to signal values in one circuit correlated with signal values in the other circuit if both circuits are given the same inputs. More precisely, our approach makes use of the fact that signal values for some signals in circuit A *imply* signal values for signals in circuit B. As a special case, this happens when the two circuits of comparison contain internal nodes that are functionally equivalent. Intuitively, the verification problem must become much easier if there is such similarity between circuits. The goal of this paper is to demonstrate the usefulness of a structural method, called *recursive learning* [14] in order to capture the similarity between circuits. Then, it is presented how this structural information can be used to reduce the complexity of functional BDD-based verification, thus exploiting the best properties of the two paradigms.

3 Review of Recursive Learning

Recursive learning allows to make precise implications [14] for a set of value assignments on a specified subset of lines in a logic circuit. Making precise implications means identifying *all* signal values that are uniquely determined as a consequence of the given situation of value assignments. That is, the precise implication procedure determines all signal values that are *necessary* for the consistency of the given situation of value assignments. To describe the procedure, the two basic notions of *unjustified gates* and *justifications* are required [14]. This paper assumes that we operate in a ternary logic alphabet (0, 1, X) where X is the don't care value. A signal is called *specified* if it is assigned the logic value 0 or 1; it is unspecified if it has the value X.

Def. 3.1: Given a gate *g* in a combinational network that has at least one specified input or output signal and the value assignments at *g* are logically consistent: *Gate g* is called *unjustified*, if there are one or several unspecified input or output signals of *g* for which there exists a combination of value assignments yielding a conflict at *g*. Otherwise, *g* is termed justified.

The special case of an unjustified gate where the specified signal is at the gate output is commonly referred to in test generation literature as an *unjustified line*. Unjustified gates describe the locations in the circuit *where* values have to be injected during the recursive learning process to examine their logic consequences; the next definition determines *what* signal values have to be injected at the unjustified gates:

Def. 3.2: Let $f_1, f_2, \ldots f_n$ be some unspecified input- or output signals of an unjustified gate g, and let $V_1, V_2, \ldots, V_n$ be logic values which specify them. The set of signal assignments, $J = \{f_1 = V_1, f_2 = V_2, \ldots f_n = V_n\}$, is called *justification* for g, if the combination of value assignments in J makes g justified.

At every unjustified gate, it is necessary to examine the logical consequences for a *complete set of justifications*:

Def. 3.3: Let ${}^gC$ be a set of m justifications $J_1, J_2, \ldots J_m$ for an unjustified gate g. If there is at least one justification $J_i \in {}^gC$, i=1,2...m for any possible justification J^* of g, such that $J_i \subseteq J^*$, then set ${}^gC$ is called *complete*.

The following represents a complete set of justifications for a logic OR gate with five unspecified inputs and a logical 1 at the output: $C = \{J_1, J_2, J_3, J_4, J_5\}$ with $J_1 = \{a = 1\}$, $J_2 = \{b = 1\}$, $J_3 = \{c = 1\}$, $J_4 = \{d = 1\}$, $J_5 = \{e = 1\}$. Note that, for example, the justification $J^* = \{a = 1, b = 0\}$ does not have to be in C, since all assignments in J_1 are contained in J^*.

These notations are used to describe a complete implication procedure in a combinational network. *Direct* implications play an important role in this reasoning. Direct implications, as in [14], refer to the evaluation of the set of value assignments at every gate that has an event, and the propagation of value assignments according to the connectivity in the circuit. Finally, the following definition is needed:

Def. 3.4: Let R be the set of value assignments $f_i = V_i$ for those variables f_i in a combinational network whose value has been changed by making implications for a given set of value assignments, S. Further, U is the set of variable assignments at the outputs of those unjustified gates which have an input with a variable assignment contained in R. The set $E(S) = R \cup U$ is called the *event list E for S*.

That is, when performing (e.g. direct) implications for a given set of value assignments S, the event list E contains all variables whose value has been changed, including output signals of new unjustified gates. The output signals of old unjustified gates are also included if their status has changed, i.e.; one of their inputs has assumed a different value. The complete implication procedure is given in Table 1. ([14] provides examples that illustrate the procedure.) A further example will be shown below.

Direct implications are performed in an event-driven way, by evaluating the truth tables of gates that have an event and by propagating the signal values according to the connectivity in the circuit. *Indirect* implications are performed by learning. What learning means is that justifications for unjustified gates are *temporarily* injected to examine their logical consequences. By performing this process recursively, it is possible to determine *all* necessary assignments. As in [14], r_{max} is a parameter which allows a reasonable tradeoff between the computational complexity and the level of precision of the implication procedure; i.e., its ability to derive necessary assignments. Note that there is a small difference between the procedure described in [14] and the recursive learning procedure described here for verification. When applying recursive learning to logic verification, indirect implications are stored as was done in the static learning procedure of [19]. Therefore, the above routine performs all direct implications along with indirect implications that have been identified and stored before.

Enumeration and reasoning in recursive learning differ greatly from conventional searching schemes for solving design automation problems. Conventional methods are based on exploring the boolean space defined by the variables of the given boolean functions. Such variable enumeration is often visualized by a decision tree or related concepts. Importantly, the search in recursive learning as illustrated in Figure 3 cannot be described by conventional enumeration schemes like decision tree based backtracking, exhaustive simulation or binary decision diagrams. In fact, the search in recursive learning can be viewed by *AND/OR trees* which are common to describe the structure in automatic theorem proving procedures for predicate logic.

Techniques in theorem proving do not completely exhaust the variable space because the variable space is usually infinite. However, they are still "refutation-complete", i.e. a wrong theorem is rejected if the problem is decidable. Note that this would not be possible if they were based on enumerating the (infinite) variable space. When dealing with boolean functions the variable space is finite and can be completely enumerated. Therefore, conventional methods in CAD can prove the satisfiability of a function by enumerating through the variable space until a set of value assignments has been found which satisfies the function, or disprove satisfiability if no such set exists. This is fundamentally different in recursive learning. The satisfiability of a function is proved if no conflict has occurred after the enumeration is completed. However, no satisfying vector has to be generated. Recursive learning is complete in the sense of refutation-completeness, i.e. it proves or disproves satisfiability but it does not generate all or any satisfying vectors. This makes it more efficient for certain applications. For example in [14] it was shown that recursive learning is far more efficient to prove the non-testability of a fault (redundancy identification) than the usual backtracking methods. In [14] it was also shown that recursive learning is complete in identifying logic implications between signal values. This is important in many applications like the one described in this paper. While the reasoning in recursive learning is similar to theorem proving techniques it is formulated such that it can be based on the same "mechanics" as traditional simulation methods or standard implication techniques in the field of design automation. For this reason, however, it should not be confused with these methods.

The search in recursive learning is illustrated by the AND/OR tree of Figure 3. It is beyond the scope of this paper to discuss the various properties of AND/OR trees constructed by the notions of recursive learning. The interested reader may refer to [22].

The procedure of Table 1 and the reasoning in recursive learning is briefly illustrated by the example shown in Figure 2. The gate whose output is z is the only unjustified gate present in the circuit. Let U denote the set of unjustified gates, let J be a justification and let C be a complete set of justifications, as defined above. No implications can be performed for the signal $z = 0$ with conventional implication routines. It is now illustrated how *make_all_implications()* as given in Table 1, identifies the necessary assignment, $c = 0$. The tree in Figure 3 schematically demonstrates the enumeration to obtain this implication. The tree in Figure 3 contains two types of nodes. To distinguish both node types, one type is marked by an arc. Nodes marked by an arc are called AND nodes and represent sets of value assignments being injected in the circuit; the other nodes are called OR nodes and represent the logic consequences derived from such sets; i.e., they represent value assignments obtained by direct implications. Value assignments belonging to unjustified gates are marked by an exclamation mark. In level 0, $\{z = 0\}$ is the initial set of value assignments, z is the only unjustified gate and $C = \{\{x1 = 0\}, \{c = 0\}\}$ is the complete set of justifications for z. The two justifications are schematically shown in level 1 of Figure 3. The assignment $x1 = 0$ is made, and direct implications are performed. This results in the value assignments $x2 = 0$ and $x3 = 0$. The new set of unjustified gates contains $x2$ and $x3$. First $x3$ is chosen. The two justifications are $e = 0$ and $d = 0$. Since the primary inputs have been reached and no new unjustified gates occur, we return to the previous level and choose unjustified gate $x2$. The two justifications are $\{x4 = 0, x5 = 0\}$ and $\{x4 = 1, x5 = 1\}$. As shown in Figure 3, both justifications lead to a conflict during their direct implications, meaning that the justification $x1 = 0$ induced by the unjustified gate z in level 0, is inconsistent. Note that this can be concluded although no inconsistency has occurred for the unjustified gate with $x3 = 0$ in level 1. This is because both unjustified gates $x3 = 0$ and $x2 = 0$ have to be consistently justified. In the graph of Figure 3 the justification $x1 = 0$ is therefore represented by an AND node. After encountering the conflict recursive learning returns into level 0 and searches the space under $c = 0$. No new unjustified gates are formed and the implications are consistent. In level 0, it is apparent that the common signal for all consistent justifications (in this case, it is only one) is $c = 0$. Hence, this becomes the implication learned $(z = 0) \Rightarrow (c = 0)$ is stored at signal z.

Similarly, just as a conventional branch-and-bound search does not actually have to build a memory-consuming decision tree, the procedure in Table 1 simply enumerates through the AND/OR tree in Figure 3 without actually building it. All operations are performed by injecting and reversing signal values, performing event-driven direct implications, and by extracting signal values if they fulfill certain conditions as given in Table 1. Therefore, the memory consumption of recursive learning, as pointed out in [14], grows linearly with the number of recursions performed.

4 A Hybrid Approach to Logic Verification

What we propose is a hybrid approach; i.e., a combination of structural and functional methods to be quite effective. Here, a structural method, recursive learning, is used to identify sub-circuits by partitioning the original circuits based on the partitioning criterion stated below. This is coupled with a functional, OBDD-based approach to prove the equivalence of these sub-circuits. Figure 4 briefly outlines the approach to be presented.

Let A and B be the circuits to be verified. Assume that both circuits are cut vertically so that A is split into A_i and A_o and B is splitted into B_i and B_o. The portion indexed "i" represents the circuit partition at the primary inputs, the portion indexed "o" represents the circuit partition at the primary outputs of the original circuit.

Criterion for circuit partitioning: A cut through circuit A and B is *permissible* if the equivalence of circuits A_o and B_o implies the equivalence of circuits A and B.

One simple means of obtaining a partitioning that fulfills this criterion is to cut in each circuit only through nodes which have an equivalent node in the other circuit, illustrated in Figure 5. It is obvious that the two circuits on the left side are equivalent. Now, consider the dashed line indicating a cut in both circuits which satisfies our criterion of circuit partitioning. This simple case demonstrates that the cut lines marked 'X' in both circuits are equivalent. The resulting sub-circuit, A_o and B_o, is shown on the right of Figure 5. Clearly, since the new pseudo-input X in circuits A_o and B_o is a functionally equivalent signal in the original circuits A and B, the equivalence of the circuits A_o and B_o implies the equivalence of the original circuits A and B. It is important to note, however, that the opposite is not true; if circuits A_o and B_o are not functionally equivalent, this does not imply the non-equivalence of A and B. As can be noted in Figure 5, functions y_1' and y_2' are not equivalent although the original circuits are. This phenomenon shall be referred to as the *false negative* problem, according to [3], and represents an unavoidable difficulty if functional and structural methods are to be combined. That is why much of what follows tackles this problem.

Note that cutting through equivalent nodes is *not* the only possibility to obtain circuit partitions that fulfill the above cutting criterion. The above cutting criterion does not require that the output nodes of A_i and B_i are equivalent. The output functions at A_i and B_i may be different as long as the difference is not observable at the outputs of the original circuits A and B. Therefore, the concept of *permissible functions* [17] is promising for identifying a larger set of possible cuts. This can be accomplished with the notions of [5] or, more generally, by recursive learning if the notion of "D-implications" as introduced in [16], is used. In the current implementation of our verification method though recursive learning operates in the usual logic alphabet $B = \{0, 1, X\}$.

Figure 4 summarizes our hybrid approach to logic verification: first a structural analysis is performed to determine a good circuit partitioning, described in detail in Section 5. Once the circuits are cut, the smaller partitioned circuits are compared for equivalence. To build OBDDs, the package described in [4] was used. Experiments suggest that false negatives are not uncommon, the reason being that the created pseudo-inputs are not always independent. The BDDs so formed may contain some combinations of pseudo-inputs which are inconsistent in the original circuit; hence they represent a don't care set for the partitioned circuit. A method to tackle this problem appears in Section 6. Note that other functional methods, e.g. [10, 20] could also be used for the functional portion of our verification tool.

5 Preprocessing: Structural Analysis by HANNIBAL

The tool HANNIBAL implements recursive learning applying knowledge about implications to *test generation* [14], *logic verification* [15] and *logic optimization* [16]. The work described here uses HANNIBAL to derive equivalent nodes in the circuits to be verified to partition the circuit. As a special case, HANNIBAL may complete the verification task alone, if it can establish the functional equivalence of the primary output nodes.

5.1 The Algorithm

Figure 6 illustrates how HANNIBAL proceeds to verify the functional equivalence of two combinational circuits, A and B. For simplicity, assume that the two circuits have only one output. Obviously, by combining the two circuits in the way shown, the logic verification problem is reduced to solving the Boolean satisfiability problem for the output signal, *e*. The circuit shown in Figure 6 has been termed *miter* by Brand [5].

In principle, the complete implication procedure of Table 1 represents a simple method to check the Boolean satisfiability of *e*: if the precise implications for $e = 1$ produce a conflict, then it follows that $e = 0$ and the two circuits are equivalent. If no conflict occurs, the precise implication procedure determines all value assignments necessary to generate a distinguishing vector. However, what maximum depth of recursion is required in order to solve the problem? As pointed out in [14], the maximum depth of recursion required to identify all necessary assignments is related to the *size* of the redundancies in the circuit. In practical circuits; i.e., circuits that realize a certain function, usually the redundant structures are rather small, so that only few recursions are needed to perform precise implications. In Figure 6 though, there is an "artificial" circuit which does not serve any practical function. If circuits A and B are functionally equivalent, then the resulting circuit represents a large redundancy and in this case, generally it seems intractable to perform precise implications.

As mentioned, recursive learning can make use of a special aspect: in many cases, "similarities" exist between the two circuits under consideration. Logically, these similarities can usually be expressed as indirect implications between signals of different circuits, schematically depicted in Figure 6. Recursive learning is a powerful technique that identifies these implications, if they exist. Note that these implications immediately indicate the functional equivalence of internal nodes (as a special case).

Figure 7 shows a flowchart of the algorithm. At first, HANNIBAL reads the description of the two circuits and combines their networks as shown in Figure 6. Essentially, the verification process consists of two phases that can be repeatedly called. In Phase 1, HANNIBAL passes through both circuits to identify and store indirect implications. At every gate, the algorithm assigns the logic signal value which makes the gate unjustified (e.g. '0' at AND) such that more than one justification exists at this gate. Note that if ‘1’ is assigned at the output of an AND gate, direct implications can be performed. Since our only interest is in storing indirect implications at gates where the direct implications “get stuck“ ‘0’ is assigned at the output of AND and NOR gates, and ‘1’ at the outputs of

OR and NAND gates. For XOR gates, both values are assigned after each other. After the assignment, *make_all_implications()* is called to perform the implications. If signal values are learned, i.e., if indirect implications exist, these implications are stored at the respective gate. This procedure is repeated for every gate in both circuits; possessing two aspects, very important for the efficiency of this preprocessing phase:

1) The gates g_i must be selected in an appropriate order. Before some gate g_i is analyzed in the described way, it must be guaranteed that all gates in the cone of influence of g_i have been treated beforehand. This ordering is necessary in order to make maximum use of prestored indirect implications.

2) In Phase 1, *make_all_implications* is not only used to learn and store indirect implications for later reference in Phase 2, but it is crucial that *make_all_implications()*, itself, makes use of previously stored indirect implications, as given in Table 1.

After Phase 1 has been completed, the actual verification algorithm is begun in Phase 2. In [15] a test generator for Phase 2 was used; this work demonstrates that the powerful concepts of OBDDs can be applied if the prestored indirect implications are used to derive a good circuit partitioning. The implications between circuits A and B immediately yield the internal equivalencies and a circuit partitioning can be derived as described in Section 3. (A detailed description of Phase 2 appears in Section 5). If the verification problem remains too complex and the base algorithm for logic verification has to abort the problem, Phase 1 is repeated with higher depth of recursion to identify more indirect implications. This continues until either a distinguishing vector is generated or the circuits are proven equivalent, or the search is aborted because a user-defined maximum recursion depth for the preprocessing phase is exceeded. Note that both, Phase 1 and 2, each represent a complete algorithm for logic verification, which complement each other.

In [13] it has been suggested to use OBDDs for identifying implications in Phase 1. This allows to exploit the power inherent to OBDDs already in Phase 1. However, if OBDDs are used to prove that certain implications are valid, additional heuristics are needed to determine a priori what nodes and values could be candidates for an implication. Further, in order to avoid building OBDDs for the full circuit the method in [13] builds OBDDs in terms of internal circuit nodes. This however neglects the interdependence between these nodes and implications can be missed. This is related to the false negative problem described in [3] and Section 4. Therefore, in order to make the learning method of [13] complete one would have to check for *each* learning operation whether an implication has been missed due to the interdependence of the BDD variables. In our approach the circuit is partitioned only a single time when switching from the structural to the functional phase. Therefore, in our approach, techniques to solve false negatives like described in Section 6 only need to be employed once after the functional comparison of Phase 2.

5.2 Examples

Example 5.1

To illustrate how HANNIBAL performs logic verification, as well as for comparison with [3], the example circuits of [3] are used, as shown in Figure 8. In Phase 1, HANNIBAL derives indirect implications between the two circuits to be compared. The procedure is as follows for the above example:

CIRCUIT 1:
Assign the appropriate value at every output signal of a gate (with at least two input signals) and use *make_all_implications(0,1)* to determine indirect implications. For simplicity, internal signal values only are listed if they belong to a gate with more than one input:

S0=1: => S2=1 => S4=1

1. justif.:   N0=0   => N0'=0

=> S0'=1

=> S4'=1

=> E=0

2. justif.:   N1=0   => N1'=0

=> S0'=1

=> S4'=1

=> E=0

we learn: S0'=1, S4'=1, E=0

indirect implications: **S0=1 => S0'=1**

(stored at S0)   **S0=1 => S4'=1, S0=1 => E=0**

S2=1: => S4=1

1. justif.:   S1=0   => S0=1

=> S0'=1(by prestored ind. implication)

=> S4'=1 (by prestored ind. implication)

=> E=0

2. justif.:   C2=0   => N2'=1

=> S4'=1, => E=0

we learn: S4'=1, E=0

indirect implications: **S2=1 => S4'=1**

(stored at S2)   **S2=1 => E=0**

S4=1:

1. justif.: S1=0 => S0=1

=> S0'=1 (by prestored ind. implication)

=> S4'=1 (by prestored ind. implication)

=> E=0

2. justif.: S3=0 => S2=1

=> S4'=1 (by prestored ind. implication)

=> E=0 (by prestored ind. implication)

we learn: S4'=1, E=0

indirect implications: **S4=1 => S4'=1**

(stored at S4) **S4=1 => E=0**

CIRCUIT 2:

Similarly, we learn the indirect implications:

S0'=1 => S0=1, S0'=1 => S2=1

S0'=1 => S4=1, S0'=1 => E=1, and

S4'=1 => S2=1, S4'=1 => S4=1, S4'=1 => E=0

OUTPUT SIGNAL:

E=1: 1. justif.: S4=1 and S4'=0

=> E=0 (by prestored ind. implication), inconsistent

2. justif.: S4=0 and S4'=1

=> E=0 (by prestored ind. implication), inconsistent

=> E=0, circuits are equivalent

In this example, Phase 1, with recursion depth 1, has been sufficient to prove that the circuits are equivalent. Illustrations of the implication procedure at higher recursion depths can be found in [14]. It is straightforward to derive the equivalent nodes from the prestored indirect implications.

Note that only one pass is performed from the inputs towards the outputs, identifying all internal equivalencies without any a priori knowledge or heuristics to identify promising candidates, representing a key advantage of the described method.

Example 5.2

It is interesting to observe that even in circuits with no structural similarity or any internal equivalencies, still it is possible to identify logic implications between signals, illustrated in Figure 9.

The upper circuit results from the lower circuit by a simple factorization step as it can occur many times during the synthesis process. It is interesting to observe that many indirect implications exist between signals in the upper and lower circuits although the two do not contain internal equivalencies and are of an entirely different structure. Recursive learning's ability to extract any type of implication between signals is an advantage over [5], significantly helping to speed up the verification process in cases where only few internal equivalencies exist.

6 Functional Comparison and Solving False Negatives

6.1 Terminology

This paper assumes familiarity with OBDDs [8]. The Ordered Binary Decision Diagrams used here are the Reduced Ordered Binary Decision Diagrams (ROBDDs) presented in [4].

- The circuit is represented as a directed acyclic graph (DAG) in the usual way, the gates being nodes in the graph. All standard limitations that apply in representing a circuit as DAG are applicable. "Node" and "gate" are used interchangeably and "BDD node" is always explicitly stated.

- A *satisfiable set* for the partitioned circuit with a single output, *y*, refers to a set of value assignments at the inputs of the partitioned circuits, A_o, (pseudo-inputs) for which *y* evaluates to 1. If these value assignments do not produce an inconsistency in the original circuit, it is called a *consistent satisfiable set*.

6.2 Functional Comparison

Figure 10 shows the flow of the verification process as in Figure 7, however with a more detailed description of phase 2. The routines *include_don't_cares()* and *consistent_satisfy()* will be described in Section 6.3.

First we need to identify the sub-circuits by partitioning the original circuits into smaller sub-circuits. For this we use a simple tracing procedure from the outputs towards the inputs. Using a depth-first search from the outputs, we trace till an equivalent node or primary input is encountered and it is marked. All these marked signals are treated as independent pseudo-inputs to the traced part of the circuit containing the outputs, i.e. these marked signals provide the desired partitions. This is performed on one circuit and the partition is mapped into the other circuit to the corresponding equivalent nodes. In some cases this mapping may not result in a complete cut in the other circuit. In these cases the outputs whose cone does not contain a complete cut is traced in a similar fashion and the cut is made complete.

The OBDDs for the outputs are calculated for each circuit in terms of these pseudo-inputs that identify the respective circuit's partition. The OBDDs are built, using the *apply* operation [4] by traversing the circuit from the pseudo-inputs towards primary outputs and building intermediate temporary OBDDs at each node's output. The OBDDs of the respective outputs are compared for equivalence. If they are isomorphic, the circuits are proved to be equivalent and if they are not isomorphic then, a new OBDD is formed by first XOR-ing the OBDDs for the respective outputs and then forming an OR of all these XOR-ed OBDDs.

6.2.1 Pseudo-input justification

The procedure is listed in Table 2. This is a process where first a satisfiable set is found by traversing the XOR-ed OBDD and next an attempt is made to find an input vector at the primary inputs of the original circuit to justify this satisfiable set. If it cannot be justified OBDD traversal is continued and a new satisfiable set is found and the process is repeated till, either it is found that there is no satisfiable set which can be justified, which means that the circuits are equivalent, or a distinguishing vector is generated.

The recursive function *consistent_satisfy()* takes an OBDD node as an argument and finds a consistent satisfiable set. This function is called with the root node of the XOR-ed OBDD. We use the same notation as used in [4] in listing our pseudo-code. If a *var* is a BDD node then, the BDD node pointed to, by the "one" branch of *var* is represented as *var.high* and the BDD node pointed to by the "zero" branch is represented as *var.low*. First, it checks the argument for a constant one or a constant zero. If it is a constant one then function *justify()* is called to justify the value assignments forming the satisfiable set, and if it cannot be justified it is not a consistent satisfiable set and the OBDD traversal is continued. A signal value one is assigned and if its implications produce inconsistency, it is erased and signal value zero is tried. If this produces a conflict, no consistent satisfiable set can be found along this path and the function returns. If the signal assignment does not produce any inconsistency, the traversal is continued by calling *consistent_satisfy* with *var.high* or *var.low* as the argument, depending on the signal assignment.

For the justification process (*justify()*) we use test generation techniques based on FAN's [11] multiple backtrace procedure and implicit enumeration. The prestored indirect implications are used to speedup the process similar to that done in [19]. Our experimental results show that *consistent_satisfy()* is generally reasonably efficient to solve the false negative problem. However, in many cases the process can be speeded up considerably by the following technique, which allows to decrease the size of the OBDD that has to be traversed by *consistent_satisfy()*.

6.2.2 Incorporating Don't Cares

We do an implication analysis to incorporate a partial information of the don't care information into the OBDDs. The procedure is listed in Table 3. As pointed out the cause of false negatives is the interdependency of the pseudo-inputs which means, if the equivalent nodes are independent then no false negatives can occur. Consider the function f of the XOR-ed partitioned circuit with n variables x_1, x_2, ..., x_n that belong to the cut in the circuit, i.e. they represent pseudo-inputs of the circuit partition for which OBDDs have been built. Function f is represented as OBDD. First f is divided into two cofactors f_i and $f_{\bar{i}}$ based on a variable $i \in \{ x_1, x_2, ..., x_n \}$. A value is assigned to the signal in one of the original (unpartitioned) circuits representing the variable i and implications are performed. (It does not matter which of the two unpartitioned circuits is taken). If this implication results in value assignments of signals, representing other pseudo-inputs, then "restrictions" are made on the respective cofactors as shown in the listing. This process is continued for all signals. Finally the "restricted" cofactors are combined by an *ITE* operator as defined in [4].

This procedure is however not complete, i.e. it does not find all don't care sets, but experimental results show that this reduces the OBDD sizes considerably.

6.3 Example

Consider the two circuits shown in Figure 11. The output functions Y_1 and Y_2 are equivalent. The ROBDD for this function is shown in Figure 12. Note that in all the examples which show OBDDs the right edges of the BDD node represent a one and its left edge represents a zero. HANNIBAL with one depth of recursion identifies the internal equivalent signals $EQ1$, $EQ2$, $EQ3$, as shown in Figure 11. Based on these internal equivalencies the circuits are partitioned and these partitioned circuits are shown in Figure 13, with their respective ROBDDs. These OBDDs are not isomorphic. An XOR of these OBDDs results in an OBDD shown in Figure 14.

Now the procedure *consistent_satisfy* is called with this XOR-ed OBDD as argument. This procedure traverses the OBDD till a terminal one is reached. In this particular example there is only one path to the terminal one, the procedure finds this path and assigns the variable values to the corresponding nodes in the original circuits. The implications of these values lead to an inconsistency in the original circuit. The traversal is continued, but since there are no more paths that lead to a one, the two circuits are proved to be equal. Note that the OBDDs for the cut circuits in Figure 13 are considerably smaller than the OBDDs of the full circuits.

7 Experimental results

In order to examine the performance of our hybrid approach we conducted a series of verification experiments on the ISCAS-85 benchmarks. The ISCAS-85 benchmarks were verified against their prime and irredundant versions [23] that are also available from MCNC. This verification experiment reflects the range of applications we have in mind for our hybrid verification method: the circuits have been modified at several different locations, but there is still “similarity“ between them. This can be expected to be the case for many practical verification problems. The goal is to show that this similarity can be identified by the structural phase 1 and be used to reduce the complexity of the functional phase 2.

The prestored indirect implications (the internal equivalent nodes) are read from a file generated by HANNIBAL [15] in phase 1. No special variable ordering techniques are used for our BDD formation. BDD variables are created for each equivalent node based on their output distance. The ordering is fixed for all the outputs of the circuits. The results are presented in Table 4 and Table 5. Table 4 compares the *final* OBDD sizes for the whole and the partitioned circuits respectively. The variables used for creating OBDDs for the whole circuit were also ordered based on their output distance. In Table 5 the CPU time in seconds is listed. The recursion depth used in phase 1 is also listed for each circuit in Table 5.

The sizes are the aggregate sizes for all the outputs taking sharing into account [4]. Importantly, in all examined cases the BDD sizes shrink drastically after the structural preprocessing phase. For some circuit marked by an * structural analysis could complete the job alone [15]. For circuit c3540 we could not build a BDD for a preprocessing recursive depth of one, so the preprocessing phase is done with a recursion depth of two. In this way, more internal equivalencies are generated which in turn makes the partitioned circuit smaller, causing the BDD sizes to shrink. This aptly demonstrates the fact that structural and functional techniques can complement each other to provide more efficient means to solve the verification problem. Note that our results can further be improved by applying more sophisticated ordering techniques as have been reported in literature. The BDD sizes for the examined circuits were very small compared to all conventional functional techniques. Consider the multiplier c6288, as was proved [8], any OBDD for a multilpier grows exponentially with the number of circuit inputs so that OBDD-based verification for c6288 is not practical. In this case, the preprocessing itself has proved that the circuits are equivalent, without a need for building an OBDD.

Table 6 lists the number of false negatives encountered for the benchmark circuit. The second column gives the number of outputs of each circuit. The number of outputs proved to be equal in the structural analysis alone are shown in the third column of this table. The fourth and the fifth columns represent the number of outputs with isomorphic OBDDs and outputs with different OBDDs (i.e. false negatives) respectively. As the results suggest the percentage of the total number of outputs that result in false negatives is fairly low. However, efficient methods have to be incorporated to effectively deal with this problem. The method presented in Section 6 performed very well for all cases of false negatives but one: the fairly large CPU-time for circuit c7552 is due to the occurrence of one out of the ten false negatives encountered for this circuit. Some fine-tuning or an improved variable ordering for this circuit may have fixed this problem, however we chose to present this result because it reflects the limitation of the hybrid approach induced by the occurrence of false negatives. Fortunately for all other cases of false negatives, also in c7552, the technique presented in Section 6 proved very efficient and contributed only little to the total CPU-time.

The results so far presented were only for the circuits which were equivalent. It is interesting to see, how our methods fare when the circuits are not equivalent. For this reason we changed a gate in one version of the benchmark circuits and verified it against the other version. Table 7 presents the results for these *true negatives* which compares OBDD sizes and CPU time between the OBDD based pure functional method and our hybrid approach. For all the cases the depth of recursion was one. In the majority of the cases the inequivalence was proved in the structural stage itself. This is because, the structural techniques are particularly powerful in generating a distinguishing vector without completely enumerating the search space. In the cases where OBDDs had to be created the required sizes were very small and in all cases the CPU time is relatively low.

Our research clearly demonstrates that functional and structural methods can be combined efficiently. The complexity of functional approaches is reduced drastically if internal equivalencies can be identified. Only in the worst case, i.e. if no internal equivalencies exist at all, the BDDs have to be constructed to their full sizes.

Conclusion - Future Work

Formal logic verification is an extremely difficult problem. No single technique is known that performs well for all classes of circuits and provides satisfactory results in all practical situations. Previous approaches to logic verification can be divided roughly into functional and structural approaches. This paper proposes a novel framework for logic verification based on the combination of structural and functional techniques. These two paradigms are of totally different nature but they can complement each other. Although the combination is generally beneficial it can create an overhead in form of *false negatives*. Summarizing, this research suggests that there are the following three ingredients to a powerful logic verification tool:

1) *Structural techniques like ATPG and recursive learning*: the task of these techniques is to capture the similarity of designs and to express them e. g. in terms of implications or internal equivalencies. This has been presented in Section 5. As pointed out in Section 4, further improvements over the approach presented here may be obtained by using observability don't cares to have a looser requirement on partitioning the circuit. This can be accomplished by the ATPG concepts of [5] or by using D-implications as in [16]. Furthermore, similarity between designs can be captured in an even more general way using the notions of [22].

2) *Graph representation of Boolean functions like OBDDs*: the task of these techniques is to formally verify circuit partitions obtained after the application of structural techniques, and to provide robustness for those cases were no or little similarity exists between designs. Further improvements over the results presented here may be obtained by better variable ordering techniques. Also, other graph representations like [10] could be employed in this context.

3) *OBDD minimization techniques using don't cares*: the task of these techniques is to handle false negatives. A method for this purpose has been presented in Section 6.2.2. Further improvements may be obtained by more sophisticated OBDD minimization techniques using don't cares like e.g.[9, 21]. Standard synthesis techniques can be applied for extracting these don't cares. Furthermore, the *compose* operation for OBDDs [4] could be used to find better cuts with smaller OBDDs when solving the false negative problem, or even compose all the way backwards in the circuit until the primary inputs are reached.

For each of these three ingredients this paper has presented a basic technique and demonstrated how it is incorporated into the general framework. It is beyond the scope of this paper to investigate all possible extensions mentioned above. Our experimental results show that the presented techniques and general framework provide a means for effective trade-off between time and memory. It is clearly demonstrated how the complexity of OBDD-based verification can be reduced by using internal, structural similarity of designs. Our framework, as we have presented, combines the advantages of the structural and functional paradigm and promises to be very useful in a logic synthesis environment.

References

[1] Aas E. J., Klingsheim K., Steen T.: "Quantifying Design Quality: A Model and Design Experiments", Proceedings of EURO ASIC 1992, pp. 172-177.

[2] Akers S.: "Binary Decision Diagrams", IEEE Transactions on Computers, vol. C-27, no. 6 June 1978, pp.509-516.

[3] Berman C. L., Trevyllian L. H.: "Functional Comparison of Logic Designs for VLSI Circuits", Intl. Conf. on Comp.-Aided Design (ICCAD), 1989, pp. 456-459.

[4] Brace K.: “Efficient Implementation of a BDD Package“, Proc. Design Automations Conference (DAC), 1990, pp. 40 -45.

[5] Brand D.: “Verification of Large Synthesized Designs”, IEEE International Conference on Computer-Aided Design (ICCAD), Nov. 1993, Santa Clara, pp. 534 - 537.

[6] Brand D. et al.: “ Incremental Synthesis“, IEEE International Conference on Computer-Aided Design (ICCAD), Nov. 1994, Santa Jose, pp. 14 - 18.

[7] Brglez F., Fujiwara H.: "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in FORTRAN", Special Session on the 1985 IEEE International Symposium on Circuits and Systems (ISCAS).

[8] Bryant R.: "Graph-Based Algorithms for Boolean Function Manipulation", IEEE Trans. on Computers, vol. C-35, no. 8, August 1986, pp. 677-691.

[9] Chang S.-C., Cheng K.T., Marek-Sadowska M.: “Minimizing ROBDD Size of Incompletely Specified Multiple Output Functions“, European Design Automation Conference (EDAC), 1994.

[10] Drechsler R. et al.: “Efficient Representation and Manipulation of Switching Functions Based on Ordered Kronecker Functional Decision Diagrams“, Proc. Design Automation Conference (DAC) 1994, pp. 415-419.

[11] Fujiwara H., Shimono T.: "On the Acceleration of Test Generation Algorithms", 13th Intl. Symp. on Fault Tolerant Comp. (FTCS), pp. 98-105, 1983.

[12] Fujita M., Kakuda T., Matsunaga Y.: “Redesign and Automatic Error Correction of Combinational Circuits“, Logic and Architecture Synthesis, ed. G. Saucier, North-Holland: Elsvier Science Publishers B.V., pp.253-262.

[13] Jain J., Mukherjee R., Fujita M.: "Advanced Verification Techniques Based on Learning", Design Automation Conference (DAC), June 1995, pp. 420 - 426.

[14] Kunz W., Pradhan D.K.: “Recursive Learning: A New Implication Technique for Efficient Solutions to CAD Problems: Test, Verification and Optimization”, IEEE Trans. on CAD, vol. 13, pp. 1143-1158, Sept. 1994.

[15] Kunz W.: "HANNIBAL: An Efficient Tool for Logic Verification Based on Recursive Learning", Proc. Intl. Conf. on Computer-Aided Design (ICCAD), Santa Clara, Nov. 1993, pp. 538-543.

[16] Kunz W., Menon P.: “Multi-Level Logic Optimization by Implication Analysis” Proc. Intl. Conference on Computer-Aided Design (ICCAD), San Jose, pp. 6-13, Nov. 1994.

[17] Muroga S. et al.: “The Transduction Method - Design of Logic Networks Based on Permissible Functions”, IEEE Trans. on Computers, Oct. 1989, pp. 1404-1424.

[18] Pomeranz I., Reddy S.M.: “On Diagnosis and Correction of Design Errors“, Proc. of Intl. Conf. on Computer-Aided Design (ICCAD), 1993, pp. 500 - 507.

[19] Schulz M., Trischler E., Sarfert T.: "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System", IEEE Trans. on CAD, vol. 7, Jan. 88, pp. 126-137.

[20] Shen A., Devadas S., Ghosh A.: “Probabilistic Construction and Manipulation of Free Boolean Diagrams”, IEEE International Conference on Computer-Aided Design (ICCAD), Nov. 1993, Santa Clara, pp. 544 - 549.

[21] Shiple T. R.: “Heursitic Minimization of BDDs Using Don’t Cares“, Proc. Design Automation Conference (DAC), 1994, pp. 225 - 231.

[22] Stoffel D., Kunz W., Gerber S.: “AND/OR Graphs“ Technical Report, Max-Planck-Society, MPI-I-95-602, 1995.

[23] Tromp G. J., van de Goor A. J.: "Logic Synthesis of 100-percent Testable Logic Networks", IEEE Intl. Conference on Computer-Design, Sept. 1991.

List of Figures

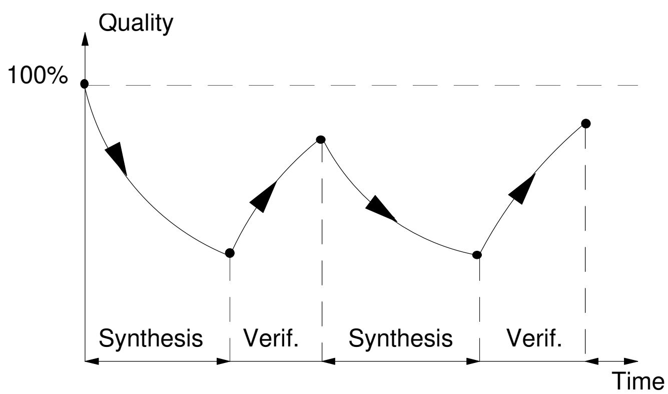

Figure 1: Typical design process (source: Aas et al., [1])

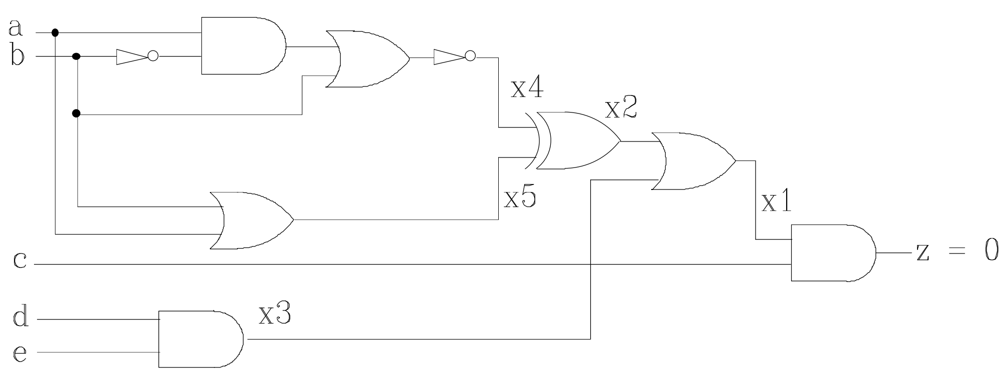

Figure 2: Circuit example to illustrate recursive learning

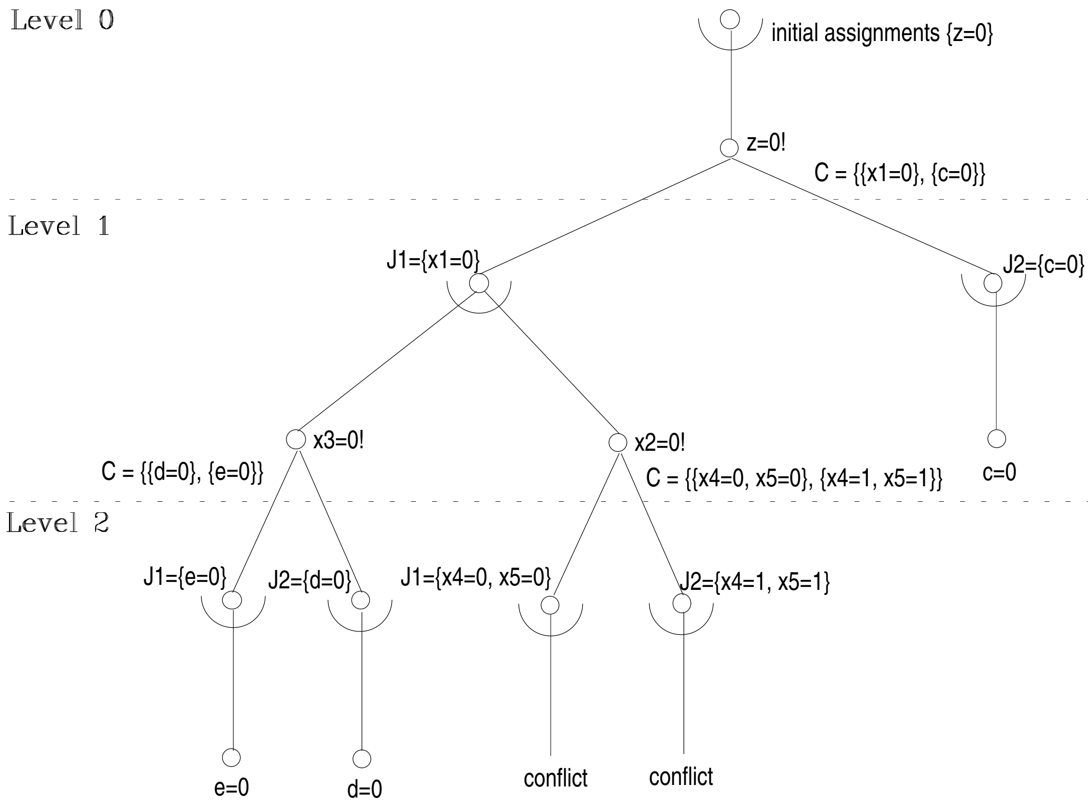

Figure 3: Enumeration to find implication z=0 ⇒ c=0

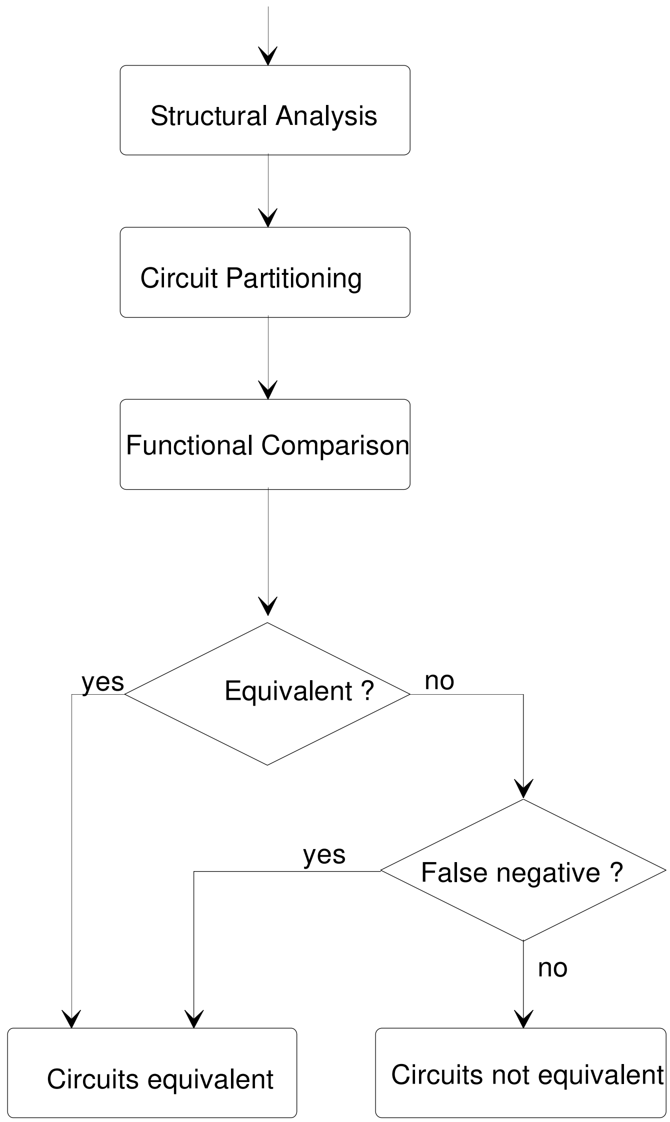

Figure 4: Hybrid structural and functional approach

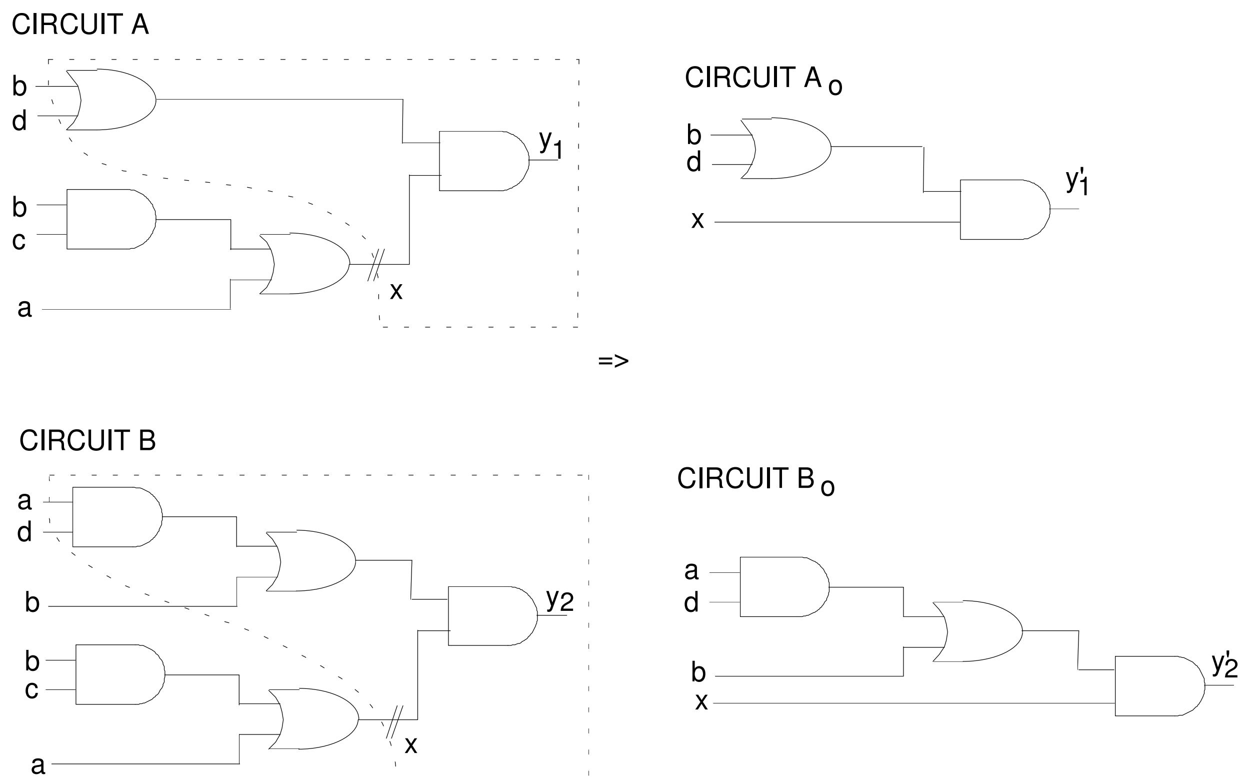

Figure 5: Circuit partitioning and false negative

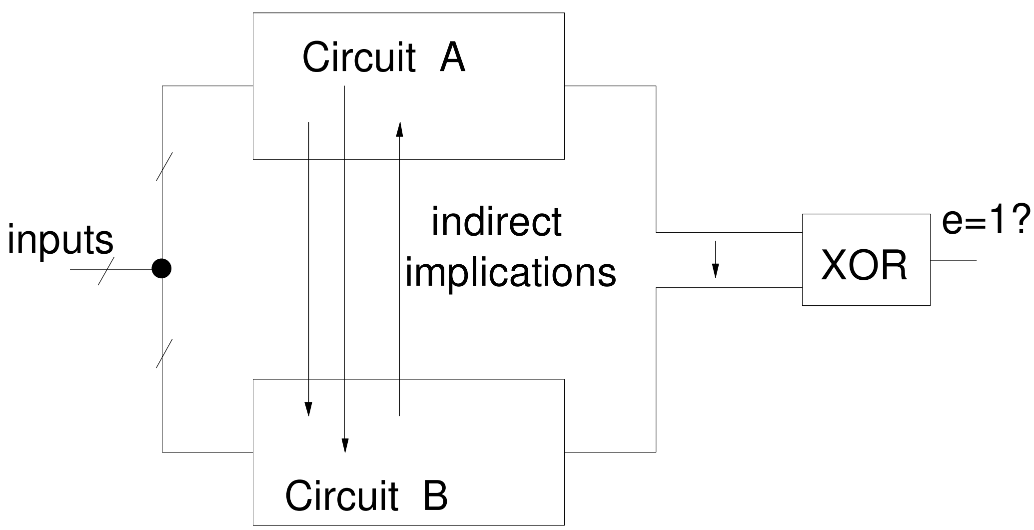

Figure 6: Logic verification by implications

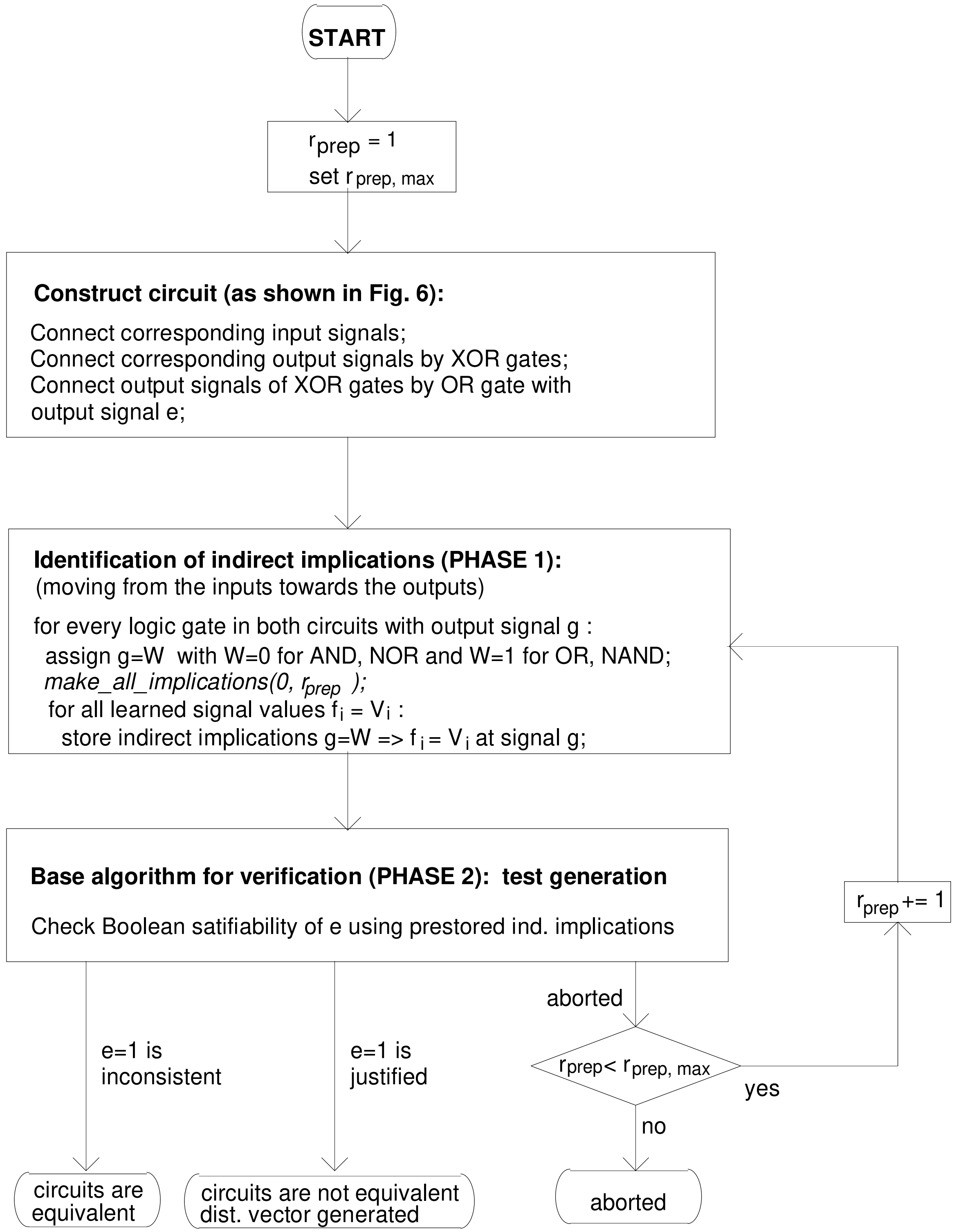

Figure 7: Flowchart of HANNIBAL (verification mode)

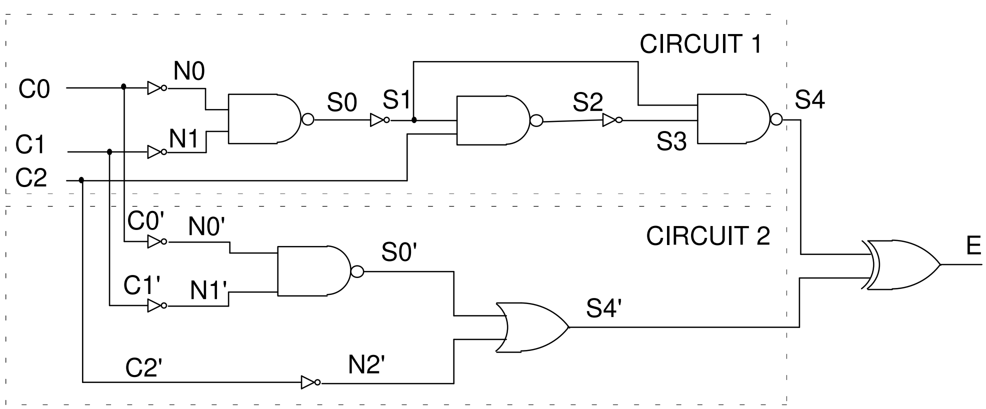

Figure 8: Example circuits [3]

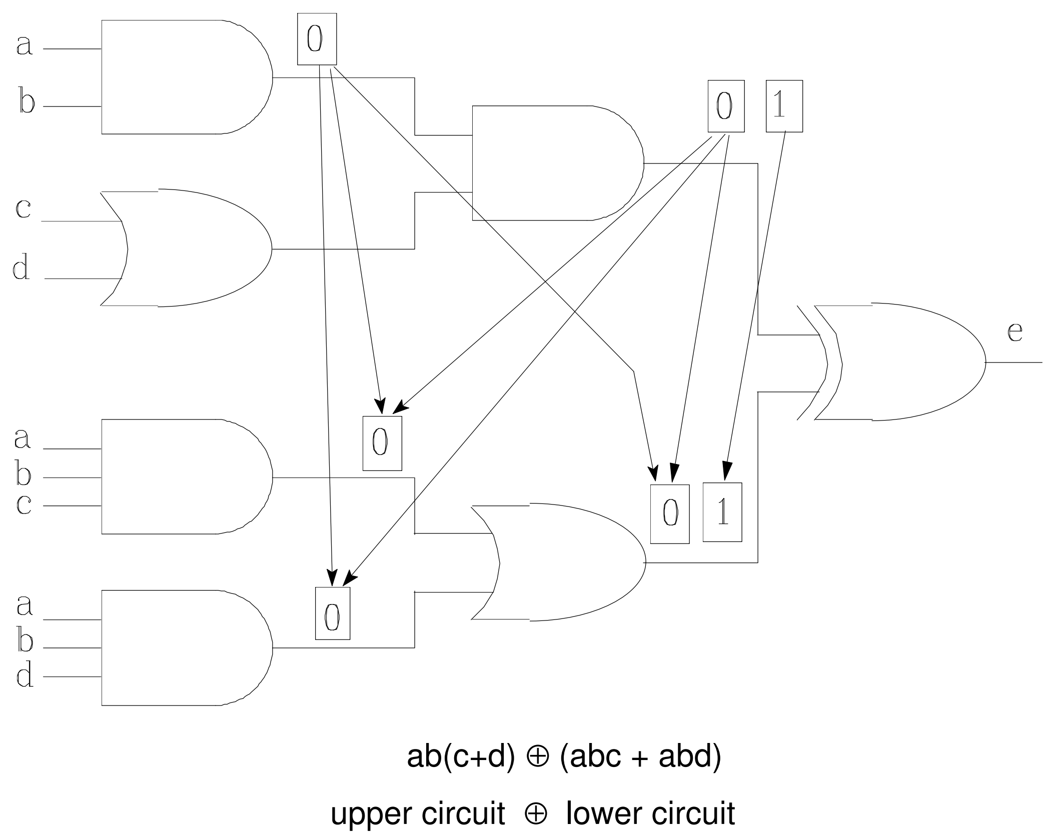

Figure 9: Indirect implications after typical synthesis operation

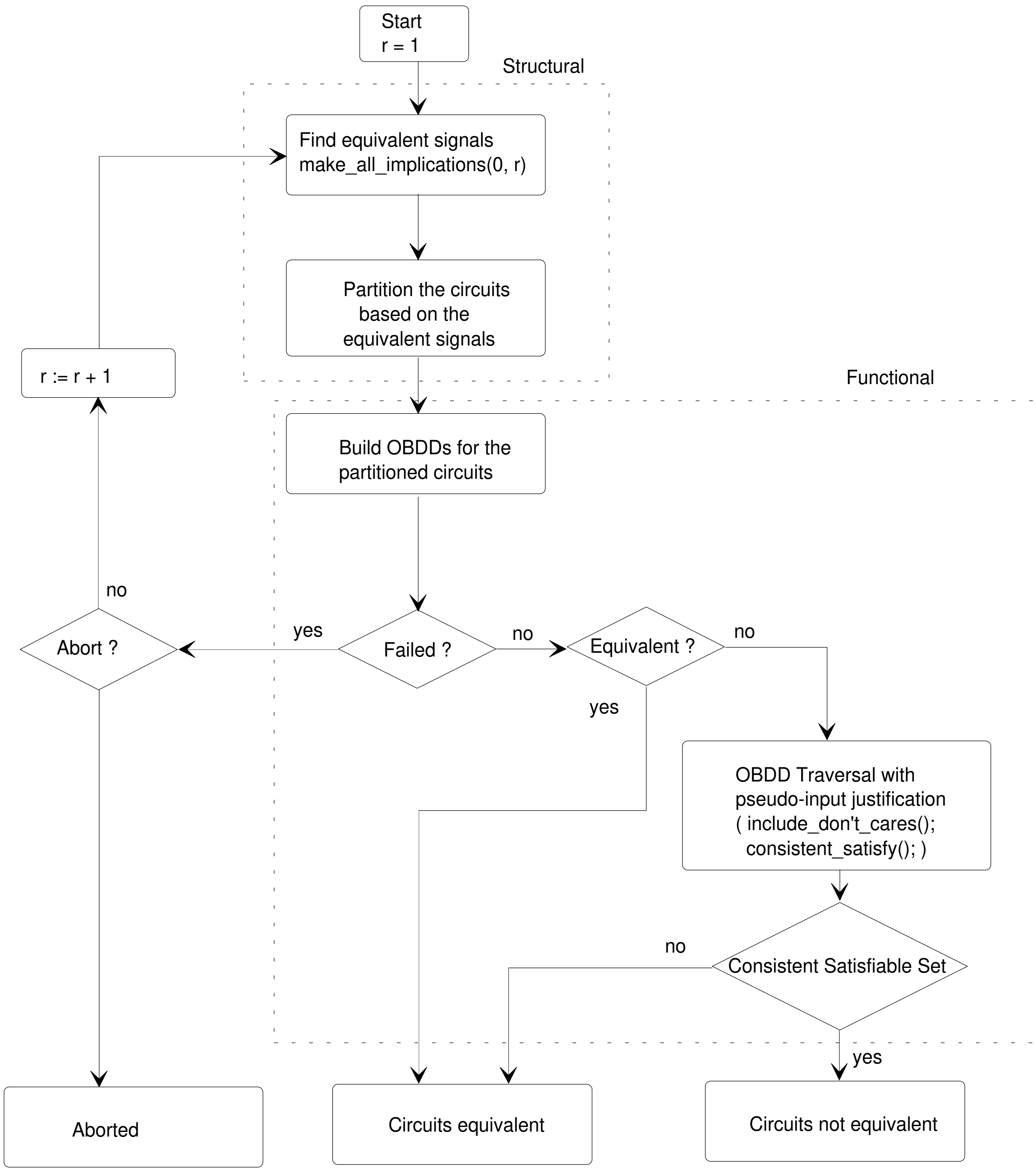

Figure 10: Block diagram of program flow in hybrid approach

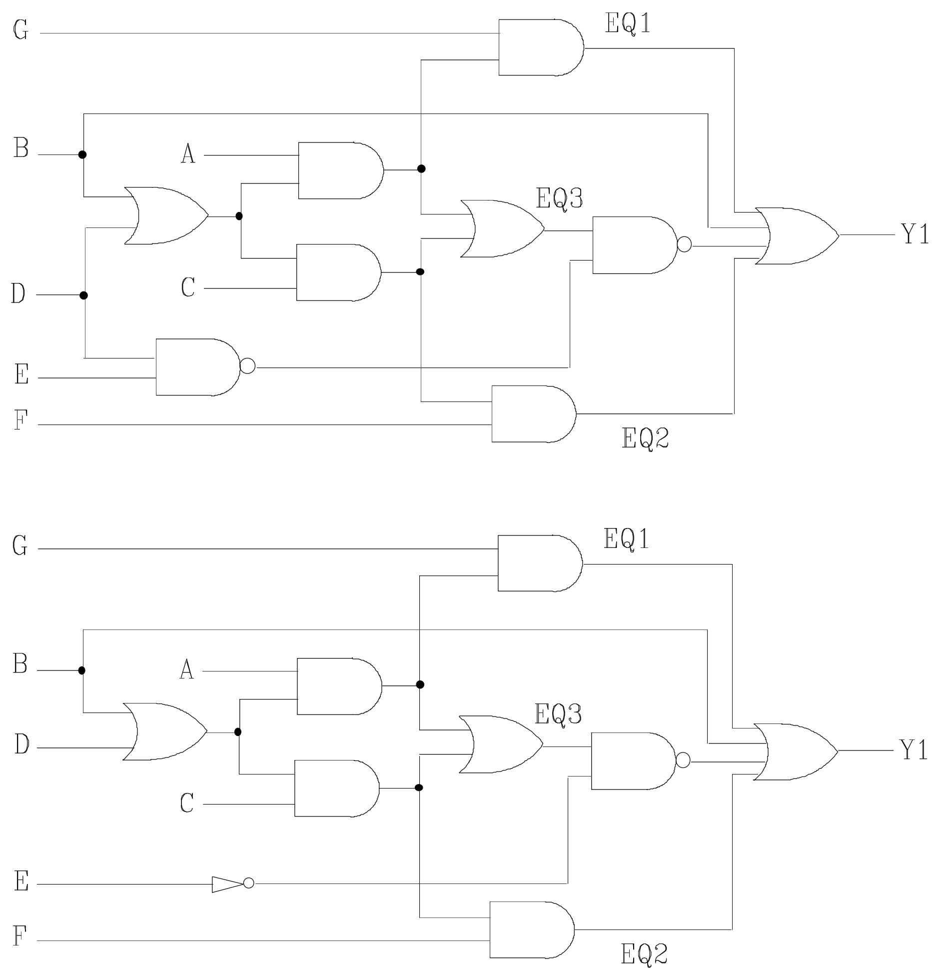

Figure 11: Example circuits (extracted from c432 [7])

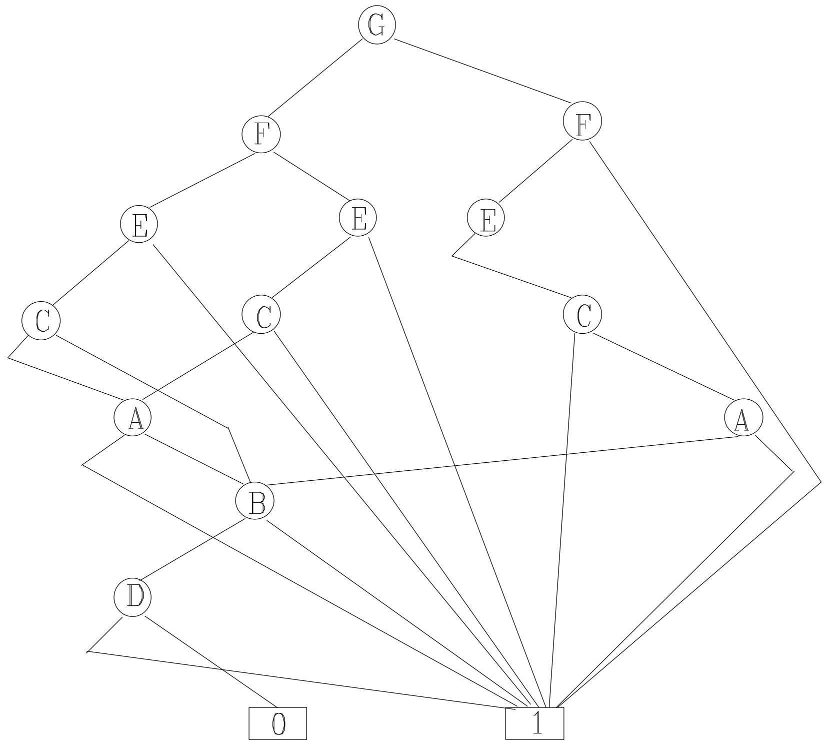

Figure 12: OBDD for circuit in Figure 11

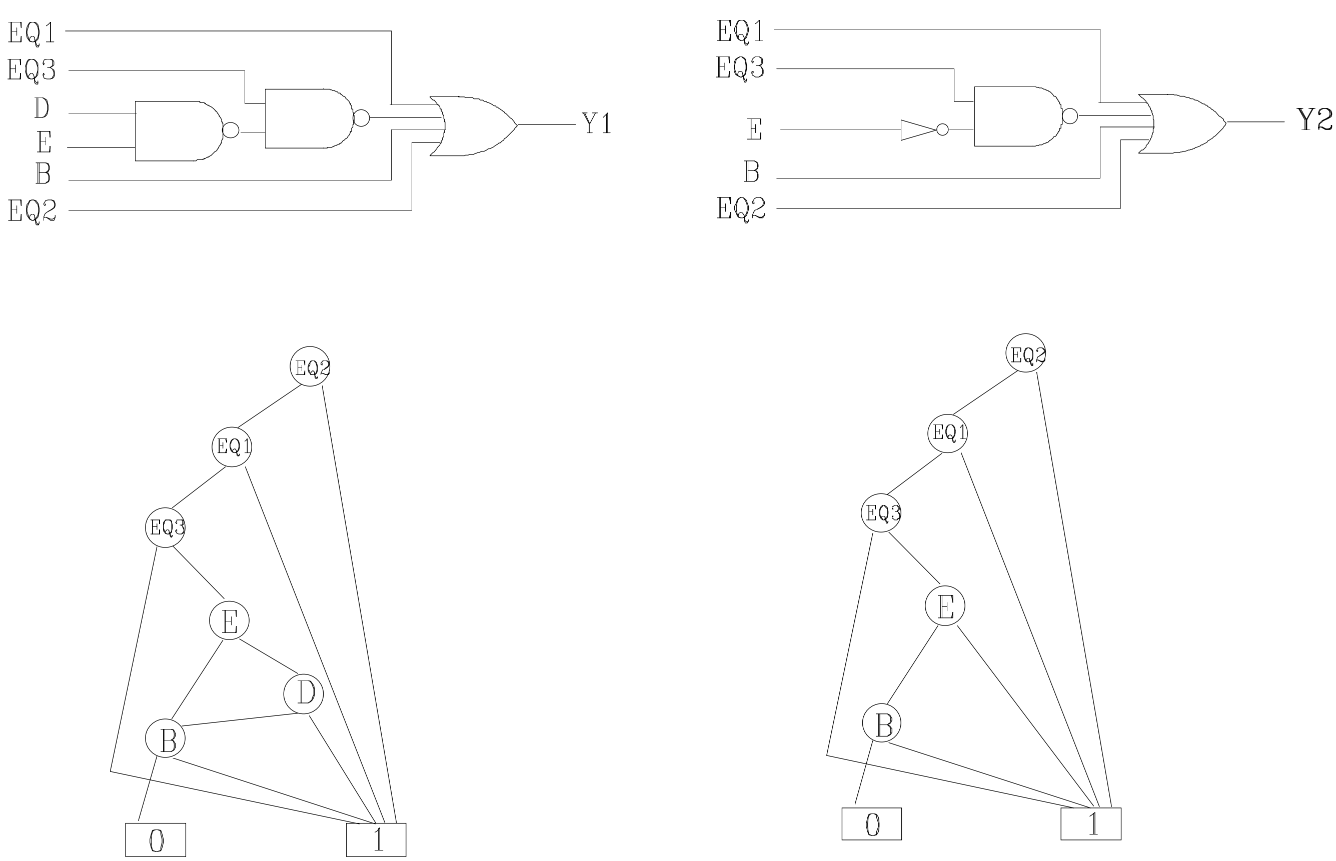

Figure 13: Partitioned circuits and their OBDDs

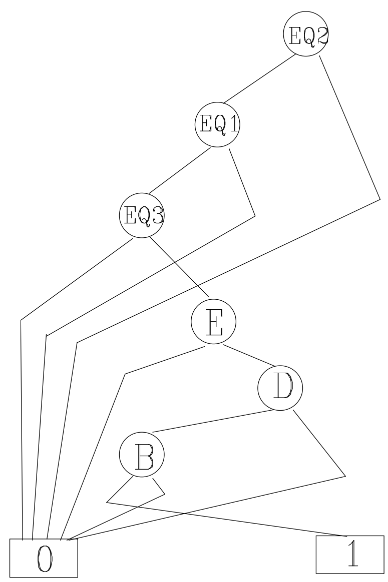

Figure 14: OBDD for XOR-ed circuit partitions (false negative)

List of Tables

```
/*    this procedure operates on a global data structure representing the circuit,
      besides r and r_max  it takes as input a set of initial value assignments S in circuit */

make_all_implications(S, r, r_max)
{
  make all direct and prestored indirect implications for S in the circuit
  and set up a list U^r of unjustified gates in event list E(S);

  if (value assignments are logically inconsistent)
      consistent := 0;
   else
      consistent := 1;

  if (r<r_max and consistent = 1)
  {
      for (each unjustified gate g in U^r)
      {
          set up list of justifications ^gC^r;
          for (each justification J_i ∈ ^gC^r )
              consistent_i := make_all_implications(J_i, r+1, r_max );

          /* extract necessary assignments */
          for (all signals f in the circuit, such that f assumes the same
              logic value V for all consistent justifications i)
          {
              /* Learning : */
              assign in the circuit : f := V;
              make all direct and prestored indirect implications in circuit and update U^r;
          }

          /* check logic consistency */
          if (consistent_i = 0 for all i)          /* all justifications are inconsistent */
          {
              consistent := 0;
              break;
          }
              else
              consistent := 1;
      }
  }
  return (consistent);
}
```

Table 1: Complete implication procedure

```
consistent_satisfy(bdd_node, var)
{
    if ( var = constant ONE)
    {
        /* found satisfiable set */
        justify(satisfiable set);
        if ( justifiable from primary inputs)
            /* found a distinguishing vector */
            return (ONE);
        else
            /* continue with the traversal */
            return (ZERO);
    }
    else if (var = constant ZERO)
        /* continue with the traversal */
        return (ZERO);

    assign:   '1' to the node in the circuit
              which the variable var represents;
    if ( imply() = consistent )
        if ( consistent_satisfy (var.high))
            return (ONE);
    erase this assignment and its implications in the circuit;

    assign:   '0' to the node in the circuit
              which the variable var represents;
    if ( imply() = consistent )
        if ( consistent_satisfy (var.low))
            return (ONE);
    erase this assignment and its implications in the circuit;
    return (ZERO);
}
```

Table 2: Routine *consistent_satisfy()*

```
/* f(x1, x2, ...,xn) is the function of the XOR-ed circuits in terms of n variables
which belong to the pseudo-inputs of the examined circuit partition */

include_don't_cares(f(x1, x2, ...,xn) represented as OBDD)
{
        for ( i = x1 to xn)
        {
            /* divide f into cofactors */
            Let v ∈ {0, 1}:
            f_i := |f|_{i = v} ;
            f_ī := |f|_{i = v̄};
            assign i := v in the original circuit;
            imply() in the original circuit;          /* make implications */
            for ( j = x1 to xn and j ≠ i )
            {
                if ( signal j in circuit assumes value w, w ∈ {0, 1} )
                    f_i := | f_i |_{j = w}
            }
            assign: i := v̄ in the original circuit;
            imply() in the original circuit;
            for ( j = x1 to xn and j ≠ i )
            {
                if ( signal j in circuit assumes value w, w ∈ {0, 1} )
                    f_ī := | f_ī |_{j = w}
            }
            f_new = ITE ( i, f_i, f_ī );
            if ( size ( f_new < f ) )
                return ( f_new );
            else
                return ( f );
        }
}
```

Table 3: Routine *include_don't_cares()*

Circuit	Functional	Structural + Functional	
	OBDD size	Rec. Depth	OBDD Size
C432	55,023	1	0*
C499	136,255	1	0*
C1355	136,255	1	0*
C1908	31,978	1	0*
C2670	unable	1	166,665
C3540	unable	2	3516
C5315	11659	1	1677
C6288	unable	1	0
C7552	unable	1	160,510

Table 4: Experimental comparison - OBDD sizes

Circuit	Functional	Structural + Functional		
	OBDD [s]	Structural [s]	Functional [s]	Total [s]
C432	60.9	1.0	0	1
C499	89.32	1.9	0	1.9
C1355	143.6	6.6	0	6.6
C1908	30.44	11.2	0	11.2
C2670	unable	8.7	150.6	159.3
C3540	unable	52.8	14.84	67.64
C5315	20.52	32.4	340.4	372.8
C6288	unable	21.5	0	21.5
C7552	unable	97.2	5486.1	5583.3

Table 5: Experimental Comparison - CPU times (Sun SPARC 10)

Circuit	Total number of outputs	# outputs found equivalent by structural analysis	# outputs with isomorphic OBDDs	# outputs with different OBDDs (False Negatives)
C432	7	7	-	-
C499	32	32	-	-
C1355	32	32	-	-
C1908	25	25	-	-
C2670	140	126	6	8
C3540	22	6	6	10
C5315	123	58	49	16
C6288	33	33	-	-
C7552	108	56	42	10

Table 6: Experimental results - false negatives

Circuit	Functional		Structural + Functional	
	OBDD size	CPU time [s]	OBDD size	CPU time [s]
C432	55047	54.01	0	1
C499	365,153	193.9	512	59
C1355	177,573	285.4	0	9
C1908	38538	23.27	0	10
C2670	unable	-	0	11
C3540	unable	-	10371	15
C5315	12889	25.29	0	26
C6288	unable	-	0	23
C7552	unable	-	0	97

Table 7: Experimental comparison - true negatives