Equivalence Checking of Dissimilar Circuits

Eugene Goldberg (Cadence Berkeley Labs, USA),
Yakov Novikov (National Academy of Science, Belarus)

Abstract

We introduce the notion of a Common Specification of circuits that generalizes the current notion of structural similarity. A CS S of circuits N_1 and N_2 is a circuit of multi-valued blocks from which N_1 and N_2 can be produced by binary encoding. We show that the equivalence checking of N_1,N_2 in general resolution (which a non-deterministic proof system) is linear in the number of blocks in S. However, there are reasons to believe that equivalence checking of circuits N_1,N_2 is hard for a deterministic algorithm if their CS is not known. We give some experimental data that substantiates this conjecture.

1. Introduction

Combinational equivalence checking is one of the most widely used techniques of formal verification. In particular, equivalence checking is the basic procedure for the verification of sequential circuits. The existing methods of equivalence checking fall into the following two classes. The first class consists of the methods that do not make any assumption about the structure of circuits to be checked for equivalence. One of the methods of this class is to compute the functionality of circuits to be checked for equivalence as BDDs [3]. Another method is to reduce equivalence checking to the satisfiability problem (SAT) and use a universal SAT-solver [4],[8]. Finally, it is possible to combine building BDDs and SAT solving as it was suggested in [5]. The main drawback of the methods of the first class is their unscalability due to exponential growth of required memory and/or runtime.

The second class consists of methods that make use of the structural similarity of circuits to be checked for equivalence. The equivalence checkers of this class are a mixture of methods using random simulation, BDDs, SAT, ATPG, graph isomorphism (see for instance [2],[6],[7]). These methods operate under the assumption that many internal points of the circuits to be compared are related by short relationships. In the majority of cases the short relationships to be deduced are equivalences [6] (sometimes implications [7] and/or replacability relationships [2]). The deduction of these relationships allows one to decompose the initial problem into a (polynomial size) set of smaller problems. Methods of this class dominate in the equivalence checking domain since they allow one to verify very large industrial designs.

In spite of the great progress achieved in equivalence checking, the existing tools can only prove the equivalence of circuits with many functionally equivalent internal points. One might say that current synthesis procedures do not change circuits very much. Therefore hard equivalence checking problems (when the number of equivalent internal points is small) are rare and should not be given any serious consideration. However, this situation may change soon. Due to close relation between synthesis and verification, a significant improvement of verification procedures may lead to putting into practice synthesis methods that would change circuits much more that they do now.

A more serious objection to a quest for more robust equivalence checkers is that the problem of equivalence checking of dissimilar circuits may be inherently hard. So no scalable equivalence checking algorithm exists for such circuits. In this paper we try to show that equvivalence checking is probably both hard and easy (and scalable) depending on whether or not the program has information about a Common Specification (CS) of the circutis to be checked for equivalence

Let N_1 and N_2 be two Boolean circuits to be checked for equivalence. A CS S is just a circuit of multi-valued gates further referred to as blocks such that N_1 (or N_2) can be obtained from S by replacing each block G of S with its implementation $I_1(G)$ (or $I_2(G)$). The circuit $I_1(G)$ (or $I_2(G)$) implements a multi-output Boolean function obtained from the truth table of G after encoding the values of multi-valued variables with binary codes.

Any pair of functionally equivalent circuits N_1,N_2 has a CS S. If N_1,N_2 are identical copies of a circuit N^*, then N^* can be considered as their CS. Each "block" of the specification N^* is "implemented" with only one gate of N_1 or N_2. So N^* is the "finest" possible CS. If N_1 and N_2 are completely structurally dissimilar, they have a trivial CS consisting of only one block where N_1 and N_2 are implementations of this block. If circuits N_1 and N_2 are structurally similar they have a set of non-trivial CSs i.e. ones different from a trivial single block CS mentioned above. (Henceforth, when we say that N_1 and N_2 have a CS we mean that they have a non-trivial CS.) The "finest" CS of N_1 and N_2 (i.e. the one with blocks of the smallest size) can be viewed as a measure of the structural similarity of these circuits. The size of a block G of S is measured in the number p of gates in the implementation of G in N_1 or N_2 (whichever is larger). If the size of the largest block of S is p we will say that S is a CS of granularity p. (Henceforth, when we say that S is a fine CS we mean that the granulrity of S is small.)

In this paper, we show that the equivalence checking of circuits N_1 and N_2 with a CS S can be performed in general resolution in $d*n*3^{6p}$ resolution steps. Here d is a constant, n is the number of blocks in S and p is the granularity of S. This result means that the complexity of equivalence checking in general resolution is linear in the number of blocks of S and exponential in their size. So, for instance, if the granularity of S is bounded by a constant, then equivalence checking is linear in the size of circuits N_1,N_2 and so is easily scalable no matter how large these circuits are. It is worth mentioning that the factor 3^{6p} is a worst case estimate for the complexity of computing so called filtering and correlation functions (see Section 4) for a block of S. In practice, computing these functions for a block should be much easier.

The upper bound on complexity of equivalence checking above is obtained in general resolution, that is in a non-deterministic proof system. However, it is not hard to formulate a deterministic algorithm for equivalence checking of circuits N_1,N_2

with a known CS S that has the same complexity as in general resolution. (This algorithm is basically formulated in Proposition 6, Proposition 7 and Proposition 8). So if one knows a fine CS of N_1 and N_2, their equivalence checking can be done very efficiently. On the other hand, there is a reason to believe that if a CS of N_1 and N_2 is unknown (even if there exists a very fine CS of N_1 and N_2), their equivalance checking is hard. This reason is that finding a short proof of equivalence basically means recovering a CS (i.e. the underlying high-level structure) from the low-level description of N_1 and N_2. It is highly unlikely that there exists an efficient algorithm for that.

To substantiate our claim that equivalence checking without any knowledge of a CS is hard we study the performance of state-of-the-art SAT-solvers Zchaff [8], BerkMin [4] and a specialized version of BerkMin called SEC (Sat based Equivalence Checking) on equivalence checking instances. All the three programs show a decent performance on MCNC benchmarks when checking the equivalence of the original and synthesized circuits. However, their perfromance quickly degrades when they are used for equivalence checking of circuits with a CS of very small granularity. In particular, some instances cannot be solved in many hours. On the other hand, an algorithm that knows the CS and generate resolution proofs described in Section 4 should be able to solve these instances in a few seconds.

2. Common Specification of Boolean Circuits

In this section, we introduce the notion of a common specification of Boolean circuits. Let S be a combinational circuit of multi-valued blocks (further referred to as a ***specification***) specified by a directed acyclic graph H. The sources and sinks of H correspond to primary inputs and outputs of S. Each non-source node of H corresponds to a multi-valued block computing a multi-valued function of multi-valued arguments. Each node of n of H is associated with a ***multi-valued variable*** V. If n is a source of H , then the corresponding variable specifies values taken by the corresponding primary input of S. If n is a non-source node of S then the corresponding variable describes the values taken by the output of the block specified by n. If n is a source (respectively a sink), then the corresponding variable is called a ***primary input variable (***respectively ***primary output variable)***. We will use the notation C=G(A,B) to indicate that a) the output of a block G is associated with a variable C; b) the function computed by the block G is G(A,B); c) only two nodes of H are connected to the node n in H and these nodes are associated with variables A and B.

Denote by $D(V)$ the ***domain*** of the variable V associated with a node of H. The value of $|D(V)|$ is called the ***multiplicity*** of V. If the multiplicity of every variable V of S is equal to 2 then S is a ***Boolean circuit***.

Now we describe how a Boolean circuit N can be produced from a specification S by encoding the multi-valued variables. Let $D(V)=\{v_1,\ldots,v_t\}$ be the domain of a variable V of S. Denote by $q(V)$ a Boolean encoding of the values of $D(V)$ that is a mapping $q{:}D(V)\rightarrow\{0,1\}^m$. Denote by $length(q(V))$ the number of bits in q that is the value of m. The value of $q(v_i)$, $v_i \in D(V)$ is called the ***code*** of v_i. Given an encoding q of length m of a variable V associated with a block of S, denote by $v(V)$ the set of m ***coding Boolean variables***.

In the following exposition we make the assumptions below.

Assumption 1. Each gate of a Boolean circuit and each block of a specification has two inputs and one output.

Assumption 2. The multiplicity of each primary input (or output) variable of a specification is a power of 2.

Assumption 3. If V is a primary input (or output) variable of a specification, then $length(q(V))=log_2(|D(V)|)$

Assumption 4. If v_1 and v_2 are values of a variable V of a specification and $v_1 \neq v_2$, then $q(v_1) \neq q(v_2)$.

Assumption 5. If A and B are two different variables of a specification , then $v(A) \cap v(B) = \emptyset$.

Remark 1. From Assumption 2, Assumption 3, and Assumption 4 it follows that if A is a primary input (or output) variable, a mapping $q{:}D(A)\rightarrow\{0,1\}^m$ is bijective. In particular, any assignment to the variables of $v(A)$ is a code of some value $a \in D(A)$.

Definition 1. Given a Boolean circuit I, denote by $Inp(I)$ (respectively $Out(I)$) the set of variables associated with primary inputs (respectively primary outputs) of I.

Definition 2. Let X_1 and X_2 be sets of Boolean variables and $X_2 \subseteq X_1$. Let y be an assignment to the variables of X_1. Denote by $proj(y,X_2)$ the ***projection*** of y on X_2 i.e. the part of y that consists of the assignments to the variables of X_2.

Definition 3. Let C=G(A,B) be a block of specification S. Let $q(A)$,$q(B)$,$q(C)$ be encodings of variables A,B, and C respectively. A Boolean circuit I is said to ***implement the block G*** if the following three conditions hold:

1) The set $Inp(I)$ is a subset of $v(A) \cup v(B)$.
2) The set $Out(I)$ is equal to v(C).
3) If the set of values assigned to $v(A)$ and $v(B)$ form codes $q(a)$ and $q(b)$ respectively where $a \in D(A)$, $b \in D(B)$, then $I(z')=q(c)$. Here z' is the projection of the assignment $z=(q(a),q(b))$ on $Inp(I)$, $I(z')$ is the value taken by I at z', and $c=G(a,b)$.

Remark 2. The reason why $Inp(I)$ may not include all the variables of $v(A)$ and/or $v(B)$ is that the function $G(A,B)$ may not distinguish some values of A or B. ($G(A,B)$ does not distinguish, say, values $a_1,a_2 \in D(A)$, if for any $b \in D(B)$, $G(a_1,b)=G(a_2,b)$.) So to implement $G(A,B)$ the circuit I may need only a subset of variables of $v(A) \cup v(B)$. This said, for the sake of simplicity, we will write $I(q(a),q(b))$ meaning $I(q'(a),q'(b))$, $q'(a)= proj(q(a),Inp(I))$ and $q'(b)=proj(q(b),Inp(I))$.

Definition 4. Let S be a multi-valued circuit. A Boolean circuit N is said to ***implement the specification S***, if it is built according to the following two rules.

1) Each block G of S is replaced with an implementation I of G.
2) Let the output of block G_1 (specified by variable R) be connected to an input of block G_2 (specified by the same variable R) in S. Then the outputs of the circuit I_1 implementing G_1 are properly connected to inputs of circuit I_2 implementing G_2. Namely, the primary output of I_1 specified by a Boolean variable $x \in v(R)$ is connected to the input of I_2 specified by the same variable of $v(R)$ if $x \in Inp(I_2)$.

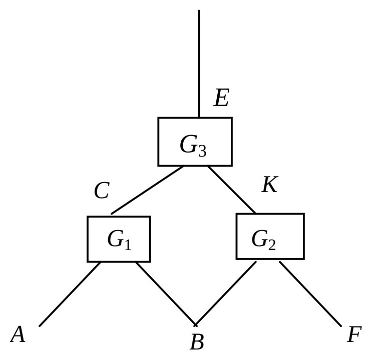
In Fig. 1*a* a specification of three blocks is shown. The functionality of two different implementations of the block

$C=G_1(A,B)$ (Fig. 1*b*) are shown in Fig. 1*c* and 1*d*. Here $D(A)=\{a_0,a_1\}$, $D(B)=\{b_0,b_1,b_2,b_3\}$ and $D(C)=\{c_0,c_1,c_2\}$. Since A and B are primary input variables they are encoded with a minimum length encoding and $q_1(A)=q_2(A)$ and $q_1(B)=q_2(B)$ where $q_1(a_0)=0$, $q_1(a_1)=1$, $q_1(b_0)=00$, $q_1(b_1)=01$, $q_1(b_2)=10$, $q_1(b_3)=11$. Finally, the encodings $q_1(C)$ and $q_2(C)$ are $q_1(c_0)=00$, $q_1(c_1)=10$, $q_1(c_2)=01$ and $q_2(c_0)=100$, $q_2(c_1)=010$, $q_2(c_2)=001$.

Remark 3. Let N be an implementation of a specification S. Let p be the largest number of gates used in an implementation of a multi-valued block of S in N. We will say that S is a specification of ***granularity*** p for N.

Definition 5. The ***topological level*** of a block G in a specification S is the length of the longest path from a primary input of S to G. (The length of a path is measured in the number of blocks on it. The topological level of a primary input is assumed to be 0.) Denote by ***level(G)*** the topological level of G in S.

Let N be an implementation of a specification S. From Assumption 4 it follows that for any value assignment h to the input variables of N there is a unique set of values $(x_1,\ldots,x_k)$, where $x_i \in D(X_i)$ such that $h=(q(x_1),\ldots,q(x_k))$. That is there is one-to-one correspondence between assignments to primary inputs of S and N. The same applies to primary outputs of S and N.



(a)

$C=G_1(A,B)$

A	B	C
a_0	b_0	c_0
a_0	b_1	c_1
a_0	b_2	c_1
a_0	b_3	c_0
a_1	b_0	c_1
a_1	b_1	c_2
a_1	b_2	c_2
a_1	b_3	c_0

(b)

$I_1(q_1(A),q_1(B))$

$q_1(A)$	$q_1(B)$		$q_1(C)$	
0	0	0	0	0
0	0	1	0	0
0	1	0	1	0
0	1	1	0	1
1	0	0	0	1
1	0	1	1	0
1	1	0	1	1
1	1	1	1	1

(c)

$I_2(q_2(A),q_2(B))$

$q_2(A)$	$q_2(B)$		$q_2(C)$		
0	0	0	1	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	1	0	0	1
1	1	0	0	0	1
1	1	1	1	0	0

(d)

Figure 1. A specification and the functionality of two implementations of a block

Definition 6. Let N be an implementation of S. Given a Boolean vector y of assignments to the primary inputs of N, the corresponding vector $Y=(x_1,..,x_k)$ such that $y=(q(x_1),\ldots,q(x_k))$ is called the ***pre-image*** of y.

Proposition 1. Let N be a circuit implementing specification S. Let $I(G)$ be the implementation of a block $C=G(A,B)$ of S in N. Let y be a value assignment to the primary input variables of N and Y be the pre-image of y. Then the values of primary outputs of $I(G)$ form the code $q(c)$ where c is the value taken by the output of G when the inputs of S take the values specified by Y.

Proof. The proposition can be proven by induction in topological levels of variables of the specification S. According to Remark 1, the proposition holds for the variables of topological level 0 (primary input variables of S). Let $C=G(A,B)$ be a block of S and $level(G)=n$, $n>0$. Let $I(G)$ be the implementation of G in N. By the induction hypothesis, values taken by the variables of $v(A)$ and $v(B)$ in N under the input assignment y should be $q(a)$ and $q(b)$ respectively. Here a and b are values of variables A and B under the input assignment Y. Then from Definition 3 it follows that the outputs of $I(G)$ take the values of $q(C)$ where $c=G(a,b)$. □

Proposition 2. Let N_1, N_2 be circuits implementing a specification S. Let each primary input (or output) variable X of S have the same encoding in N_1 and N_2. Then Boolean circuits N_1 and N_2 are functionally equivalent.

Proof. Let y be an arbitrary assignment to input variables of N_1 and N_2. Since the encodings of primary input variables of S in N_1 and N_2 are the same, then the pre-image Y of y for N_1 and N_2 is the same. Let C be a primary output variable of S associated with a block G. From Proposition 1 it follows that the values taken by the implementations $I_1(G)$ and $I_2(G)$ of G in N_1 and N_2 are equal to $q_1(c)$ and $q_2(c)$ respectively. Here c is the value taken by the output of G under input assignment Y and q_1 and q_2 are encodings of the primary output variable C in N_1 and N_2. Since C has the same encoding in N_1 and N_2, then $q_1(c) = q_2(c)$. □

Definition 7. Let N_1, N_2 be two functionally equivalent Boolean circuits. Let N_1, N_2 implement a specification S so that for every primary input (output) variable X encodings $q_1(X)$ and $q_2(X)$ (used when producing N_1 and N_2 respectively) are identical. Then N is called ***a common specification*** (CS) of N_1 and N_2.

Remark 4. Let S be a CS of N_1,N_2 and C be a variable of S. We will assume that $v_1(C) = v_2(C)$ if C is a primary input variable and $v_1(C) \cap v_2(C) = \varnothing$ otherwise.

Definition 8. Let S be a CS of N_1,N_2. Let p_1 (respectively p_2) be the granularity of S with respect to N_1 (respectively N_2). Then we will say that S is a CS of N_1,N_2 of ***granularity*** $p = max(p_1,p_2)$.

Definition 9. Given two functionally equivalent Boolean circuits N_1, N_2, S is called the ***finest common specification*** if it has the smallest granularity p among all the CSs of N_1 and N_2.

3. Equivalence Checking as SAT

Since in this paper we formulate the complexity of equivalence checking in terms of resolution proofs, we recall a common way of reducing equivalence checking to the satisfiability problem.

Definition 10. A disjunction of literals of Boolean variables not containing two literals of the same variable is called a ***clause***. A conjunction of clauses is called a ***conjunctive normal form*** (CNF).

Definition 11. Given a CNF F, ***the satisfiability problem*** (SAT) is to find a value assignment to the variables of F for which F evaluates to 1 (also called a ***satisfying assignment***) or to prove that such an assignment does not exist. A clause K of F is said to be ***satisfied*** by a value assignment y if $K(y)$=1.

The standard conversion of an equivalence checking problem into an instance of SAT is performed in two steps. Let N_1 and N_2 be Boolean circuits to be checked for equivalence. At the first step of this conversion, a circuit M called ***a miter*** [2] is formed from N_1 and N_2. The miter M is obtained by 1) identifying the corresponding primary inputs of N_1 and N_2; 2) XORing each pair of corresponding primary outputs of N_1 and N_2; 3) ORing the outputs of the added XOR gates. So the miter of N_1 and N_2 evaluates to 1 if and only if for some input assignment a primary output of N_1 and the corresponding output of N_2 evaluate to different values. Therefore, the problem of checking the equivalence of N_1 and N_2 is equivalent to testing the satisfiability of the miter of N_1 and N_2.

At the second step of conversion, the satisfiability of the miter is reduced to that of a CNF formula F. This formula is a conjunction of CNF formulas F_1,..,F_n specifying the functionality of the gates of M and a one-literal clause that is satisfied only if the output of M is set to 1. The CNF F_i specifies the i-th gate g_i of M. Any assignment to the variables of F_i that is inconsistent with the functionality of g_i falsifies a clause of F_i (and vice versa, a consistent assignment satisfies all the clauses of F_i.) For instance, the AND gate $y=x_1x_2$ is specified by the following three clauses $\sim x_1 \vee \sim x_2 \vee y$, $x_1 \vee \sim y$, $x_2 \vee \sim y$.

4. Equivalence Checking in General Resolution

In this section, we prove some results about the complexity of equivalence checking of circuits with a CS of granularity p. The main idea of the proof is that if S is a CS of N_1 and N_2, then their equivalence checking reduces to computing filtering and correlation functions for each variable of S. The two main properties of these functions are that

- They can be built based only on the information about the topology of S and about "assignment" of gates of N_1 and N_2 to blocks of S.
- Filtering and correlation functions for a variable C specifying the output of a block $G(A,B)$ can be computed "locally" from filtering and correlation functions of variables A and B and CNFs specifying implementations $I_1(G)$ and $I_2(G)$. So these functions can be computed in topological order starting with inputs and proceeding to outputs.

Definition 12. Given a constant p, a CNF formula F is a member of the ***class M(p)*** if and only if it satisfies the following two conditions.

- F is the CNF formula (obtained by the procedure described in Section 3) specifying the miter of a pair of functionally equivalent circuits N_1,N_2.
- N_1,N_2 has a CS of granularity p.

Definition 13. Let K and K' be clauses having opposite literals of a variable (say variable x) and there is only one such variable. The ***resolvent*** of K , K' in variable x is the clause that contains all the literals of K and K' but the positive (i.e. literal x) and negative (i.e. literal $\sim x$) literals of x. The operation of producing the resolvent of K and K' is called ***resolution***.

Definition 14. ***General resolution*** is a proof system of propositional logic that has only one inference rule. This rule is to resolve two existing clauses to produce a new one. Given a CNF formula F, a proof $L(F)$ of unsatisfiability of F in the general resolution system consists of a sequence of resolutions resulting in the derivation of an ***empty clause*** (i.e. a clause without literals).

General resolution is complete, which means that given an unsatisfiable formula F there is always a proof $L(F)$ that derives an empty clause.

Definition 15. Let F be a set of clauses. Denote by ***supp(F)*** the set of variables whose literals occur in clauses of F.

The following three propositions are used in the proof of Proposition 8.

Proposition 3. Let F be a set of clauses that implies a clause K. Then there is a sequence of resolutions of at most $3^{|supp(F)|}$ steps that results in the derivation of a clause that implies K.

Proof. Denote by F' the formula that is obtained from F by making the assignments that set the literals of K to 0 (and removing the satisfied clauses and the literals set to 0). It is not hard to see that F' is unsatisfiable since it implies an empty clause. So there is a resolution proof $L(F')$ that results in deducing an empty clause. Then by replacing each clause of F' involved in $L(F')$ with its "parent" clause from F we get a sequence of resolutions resulting in deducing a clause that implies K. The number of resolvents in $L(F')$ cannot be more than $3^{|supp(F')|}$ (i.e. the total number of clauses of $|supp(F')|$ variables) and so it cannot be more than $3^{|supp(F)|}$. □

Proposition 4. Let A,B,C be Boolean functions and $A \wedge B \rightarrow C$. Then for any function $A' \rightarrow A$, it is true that $A' \wedge B \rightarrow C$.

Proof. Let x be an assignment that sets $A' \wedge B$ to 1. Then $A'(x)$=1 and $B(x)$=1. Since $A' \rightarrow A$, then $A(x)$=1. Then $A(x) \wedge B(x)$ = 1 and so $C(x)$=1.

Proposition 5. Let X_1 and X_2 be sets of Boolean variables, $F(X_1,X_2)$ and $H(X_2)$ be CNF formulas and F imply H. Then in at most $3^{|supp(F)|}$ resolution steps one can derive a CNF formula H' that implies $H(X_2)$ such that $supp(H') \subseteq supp(H)$.

Proof. Let K be a clause of H. From Proposition 3 it follows that in at most $3^{|supp(F)|}$ steps one can derive a clause K' that implies K. Since $K' \rightarrow K$, then $supp(K') \subseteq supp(K)$. So in at most $|H| * 3^{|supp(F)|}$ steps, where $|H|$ is the number of clauses in H, one can derive a CNF H' implying H such that $supp(H') \subseteq supp(H)$. (The fact that H' implies H follows from Proposition 4.) However, if one does not produce the same resolvent twice, the total number of resolution steps when deriving H' cannot be more than $3^{|supp(F)|}$ (because it is the total number of clauses of $|supp(F)|$ variables).

Definition 16. Let N be an implementation of a specification S. Let C be a variable of S. A function Ff is called ***a filtering function*** if:

- supp(Ff) $\subseteq$ $v(C)$.
- If an assignment z to the variables of $v(C)$ is a code $q(c)$, $c \in D(C)$, then $Ff(z)$=1. Otherwise, $Ff(z)$=0.

Remark 5. If C is a primary input variable of S , then $Ff(v(C))\equiv 1$. Indeed, as it follows from Remark 1 any assignment to C is the code of a value $c \in D(C)$.

Proposition 6. Let N be an implementation of a specification S. Let $C=G(A,B)$ be a block of S. Let F be the CNF formula specifying N built as described in Section 3 and $F(I(G))$ be the part of F specifying the implementation $I(G)$ of G in N. Then $P \rightarrow Ff(v(C))$ where $P=Ff(v(A)) \wedge Ff(v(B)) \wedge F(I(G))$.

Proof. To prove that $P \rightarrow Ff(v(C))$ one needs to show that any assignment that sets P to 1 also sets $Ff(v(C))$ to 1. It is not hard to see that the support of all the functions of the expression $P \rightarrow Ff(v(C))$ is a subset of $supp(F(I(G)))$. Let $h=(x,y,z)$ be an assignment that sets P to 1 where x,y,z are assignments to the variables from $v(A)$,$v(B)$ and $v(C)$ respectively. Then h has to set to 1 the functions $Ff(v(A))$, $Ff(v(B)$, $F(I(G))$. Since h sets $Ff(v(A))$ to 1, then $x=q(a)$, $a \in D(A)$. Since h sets $Ff(v(B))$ to 1, then $y=q(b)$, $b \in D(B)$. So $h=(q(a),q(b),z)$. To set to 1 $F(I(G))$ assignment z has to be equal to $q(c)$, where $c=G(a,b)$. Then h sets $Ff(v(C))$ to 1. □

Definition 17. Let S be a CS of circuits N_1 and N_2 and C be a variable of S. A function Cf is called ***a correlation function*** for encodings q_1 and q_2 of the values of C (used when producing N_1 and N_2) if :

- $supp(Cf) \subseteq v_1(C) \cup v_2(C)$.
- $Cf(z_1, z_2)=1$ for any assignment z_1 to $v_1(C)$ and z_2 to $v_2(C)$ such that $z_1=q_1(c)$ and $z_2=q_2(c)$ where $c \in D(C)$. Otherwise $Cf(z_1, z_2)=0$.

Remark 6. If C is a primary input variable of S, then $Cf(v_1(C),v_2(C)) \equiv 1$. Indeed, as it follows from Remark 1 any assignment to $v_1(C)$ or $v_2(C)$ is the code of a value $c \in D(C)$. Besides, from the definition of CS it follows that $q_1(C)=q_2(C)$. Finally, from Remark 4 it follows that $v_1(C)=v_2(C)$. So any assignment (x,y) to the variables of $v_1(C)$,$v_2(C)$ can be represented as $(q_1(c),q_2(c))$, $c \in D(C)$.

Proposition 7. Let S be a CS of circuits N_1,N_2. Let $C=G(A,B)$ be a block of S. Let F be the CNF formula specifying the miter of N_1,N_2 built as described in Section 3. Let $F(I_1(G))$ and $F(I_2(G))$ be the part of F specifying the implementation $I_1(G)$ and $I_2(G)$ of G in N_1 and N_2 respectively. Then P implies $Cf(v_1(C),v_2(C))$. Here P = $Filtering \wedge Correlation \wedge Implementation$ and $Filtering = Ff(v_1(A)) \wedge Ff(v_1(B)) \wedge Ff(v_2(A)) \wedge Ff(v_2(B))$, $Correlation = Cf(v_1(A),v_2(A)) \wedge Cf(v_1(B),v_2(B))$, $Implementation = F(I_1(G)) \wedge F(I_2(G))$.

Proof. To prove that P implies $Cf(v_1(C),v_2(C))$ one needs to show that any assignment that sets P to 1 also sets $Cf(v_1(C),v_2(C))$ to 1. It is not hard to see that the support of all the functions of the expression $P \rightarrow Cf(v_1(C),v_2(C))$ is a subset of $supp(F(I_1(G)) \cup supp(F(I_2(G))$. Let $h=(x_1, x_2, y_1, y_2, z_1, z_2)$ be an assignment that sets P to 1 where $x_1, x_2, y_1, y_2, z_1, z_2$ are assignments to $v_1(A)$, $v_2(A)$, $v_1(B)$, $v_2(B)$, $v_1(C)$, $v_2(C)$ respectively. Then h has to set to 1 all the functions the conjunction of which forms P. Since h has to set the function $Filtering$ to 1, then $x_1=q_1(a_1)$, $x_2=q_2(a_2)$ where $a_1,a_2 \in D(A)$ and $y_1=q_1(b_1)$, $y_2=q_2(b_2)$, where $b_1,b_2 \in D(B)$. So $h=(q_1(a_1),q_2(a_2), q_1(b_1),q_2(b_2), z_1, z_2)$. Since h sets the function $Correlation$ to 1 then a_1 has to be equal to a_2 and b_1 has to be equal to b_2. So h can be represented as $(q_1(a),q_2(a), q_1(b),q_2(b), z_1, z_2)$ where $a \in D(A)$ and $b \in D(B)$. Since h sets the function $Implementation$ to 1, then z_1 has to be equal to $q_1(c)$, $c=G(a,b)$ and z_2 has to be equal to $q_2(c)$. So h is equal to $(q_1(a),q_2(a),q_1(b),q_2(b),q_1(c),q_2(c))$ and hence it sets the correlation function $Cf(v_1(C),v_2(C))$ to 1. □

Proposition 8. Let F be a formula of $M(p)$ specifying the miter of circuits N_1,N_2 obtained from a CS S of granularity p. The unsatisfiability of F can be proven by a resolution proof of no more than $d*n*3^{6p}$ resolution steps where n is the number of blocks in S and d is a constant.

Proof. From Proposition 6 and Proposition 7 it follows that one can deduce correlation and filtering functions for all the variables of S starting with blocks of topological level 1 and proceeding in topological order. Indeed, let $C=G(A,B)$ be a block of topological level 1. Then A and B are primary input variables and the filtering and correlation functions for them are known (they are tautologies). From Proposition 6 it follows that $Ff(v_1(C))$ and $Ff(v_2(C))$ are implied by $F(I_1(G))$ and $F(I_2(G))$ respectively. From Proposition 5 it follows that a CNF implying $Ff(v_1(C))$ (respectively $Ff(v_2(C))$) can be derived by resolving clauses of $F(I_1(G))$ (respectively $F(I_2(G))$). Similarly, the correlation function $Cf(v_1(C),v_2(C))$ is implied by $F(I_1(G)) \wedge F(I_2(G))$. So a funciton implying $Cf(v_1(C),v_2(C))$ can be derived from the latter by resolution. From Proposition 4 it follows that to apply Proposition 6 and Proposition 7, instead of the functions $Ff(v_1(C))$, $Ff(v_2(C))$ and $Cf(v_1(C),v_2(C))$, one can use any functions implying them. After filtering and correlation functions are computed for all the variables of level 1, the same procedure can be applied to variables of topological level 2 and so on. If S consists of n blocks, then in n steps one can deduce correlation functions for the primary output variables of S. At each step two filtering and one correlation function are computed for a variable $C=G(A,B)$ of S. The complexity of this step is no more than 3^{6p}. Indeed, the support of all functions mentioned in Proposition 6 and Proposition 7 needed for computing $Ff(v_1(C))$, $Ff(v_2(C))$ and $Cf(v_1(C),v_2(C))$ is a subset of $A=supp(F(I_1(G))) \cup supp(F(I_2(G)))$. The total number of gates in $I_1(G)$ and $I_2(G)$ is bounded by $2p$, each gate having 2 inputs and 1 output. So the total number of variables in A cannot be more than $6p$. Then from Proposition 5 it follows that in at most 3^{6p} steps one can deduce CNFs implying $Ff(v_1(C))$, $Ff(v_2(C))$ and $Cf(v_1(C),v_2(C))$. Then the total number of resolution steps one needs to deduce functions implying the correlation functions for the primary output variables of S is bounded by $n*3^{6p}$.

Now we show that from the correlation functions for primary output variables of S, one can deduce an empty clause in the number of resolution steps linear in $n*p$. Let C be a primary output variable specifying the output of a block G of N. Let $I_1(G)$ and $I_2(G)$ be the implementations of G in N_1 and N_2 respectively. Let $|D(C)|=2^k$ (By Assumption 2 the multiplicity of C is a power of 2.) Then $length(q_1(C))= length(q_2(C))=k$. (By Assumption 3, values of S are encoded by a minimal length encoding.)

Now we show that there is always a correlation function $Cf(v_1(C),v_2(C))$ that implies the CNF consisting of k pairs of two literal clauses specifying the equivalence of corresponding outputs of $I_1(G)$ and $I_2(G)$. Let f_1 and f_2 be two Boolean variables of $v_1(C)$ and $v_2(C)$ respectively that specify corresponding outputs of N_1 and N_2. Since S is a CS of N_1 and N_2, then $q_1(C)=q_2(C)$. So any assignment $q_1(c)$,$q_2(c)$ to v_1(C) and v_2(C) that satisfies $Cf(v_1(\mathrm{C}),v_2(\mathrm{C}))$ also satisfies clauses $K'=f_1 \vee \sim f_2$ and $K''=\sim f_1 \vee f_2$. So K' and K'' are implied by $Cf(v_1(C),v_2(C))$ and so clauses implying them can be deduced by the procedure described in the proof of Proposition 7. (The resolution steps one needs to deduce equivalence clauses are already counted in the expression $n{*}3^{6p}$)

Using each pair of equivalence clauses K' and K'' (or clauses implying them) and the clauses specifying the gate g=XOR(f_1,f_2) of the miter, one can deduce a single literal clause ~g. This clause requires setting the output of this XOR gate to 0. Each such a clause can be deduced in the number of resolutions bounded by a constant and the total number of such clauses cannot be more than $n{*}p$. Finally, from these unit clauses and the clauses specifying the final OR gate of the miter, the empty clause can be deduced in the number of resolutions bounded by $n{*}p$. So the empty clause is deduced in no more than $n{*}3^{6p} + d'{*}n{*}p$ steps where d' is a constant. Finally, one can pick a constant d such $n{*}3^{6p} + d'{*}n{*}p \leq d{*}n{*}3^{6p}$ □

Remark 7. In Proposition 8 we give a worst case estimate of the complexity of deducing filtering and correlation functions. In practice, this complexity can be much lower. In a sense, the best way to interpret the theory developed in this section is that the complexity of equivalence checking of circuits N_1,N_2 with a CS S is linear in the number of blocks in S.

5. Tuning a General Purpose SAT-Solver for Equivalence Checking

In Section 4 we showed that equivalence checking of circutis with a fine CS is easy in general resolution. It can be also shown that there is a deterministic algorithm of equivalence checking of circuits N_1,N_2 with a known CS S whose complexity is the same as in general resolution. In particular, this algorithm is linear in the number of blocks in S and exponential in their size. (The description of this algorithm will be included in our next paper.)

Now we want to show that the equivalence checking of circuits having a CS is hard if the latter is not known. The theory that could answer this question is still in development [9]. So we made an attempt to substantiate our claim experimentally. The list of tools that it makes sense to try for equivalence checking of circuits with a CS is very short. Regular methods of equivalence checking that are based on establishing strong relationships (like equivalence and/or implication) between internal points of N_1 and N_2 are useless. It is very easy to generate ciruits with a fine CS that have neither equivalences nor implications between internal points of N_1 and N_2.

Probably, the most reasonable thing to do is to try state-of-the-art universal SAT-solvers like [4][8]. On the one hand, they show good performance on many classes of CNF formulas. On the other hand, modern SAT-sovlers are very efficient and can handle quite large instances. The main drawback of a universal SAT-solver is that it does not make any assumptions about the structure of the formula. To solve this problem we developed a Sat-based Equivalence Checker (SEC) that is a modification of our SAT-solver BerkMin. In contrast to BerkMin, SEC has some knowledge about the topology of circuits to be checked for equivalence (but it does not have any information about CSs).

The detailed description of BerkMin can be found in [4]. For the lack of space, here we describe only the features of BerkMin that has been changed in SEC. BerkMin is based on conflict clause recording. A conflict clause "encodes" a contradictory assignment to variables of the formula that implies assigning opposite values to some variable. Such a conflict occurs at each leaf of the search tree built by the SAT-solver. After encountering such a conflict the SAT-solver has to backtrack. A conflict clause is recorded to avoid repeating the same contradictory assignment again.

Conflict clauses are stored in BerkMin as a chronologically ordered stack that plays a key role in BerkMin's decision-making. (The most recently deduced clause is put on the top of the stack.). Namely, the next variable to be branched on is selected among the variables whose literals are in the topmost unsatisfied clause K of the stack. (If all the conflict clauses are satisfied and only some clauses of the initial formula are left unsatisfied, SEC uses the same decision making procedure as BerkMin.) Among the variables of K, the one with the highest activity is selected. The activity of a variable x is the number of times clauses containing either literal of x have been involved in conflicts. The activity of each variable is periodically divided by a small constant as it was suggested in [8].

One more feature of BerkMin is its use of restarts. From time to time BerkMin abandons the current search tree (preserving deduced conflict clauses) and starts building a new one. Before the restart BerkMin runs a data-base cleaning procedure to get rid of clauses that are not used in conflicts any more and clauses that have become redundant due to deducing single literal conflict clauses.

In SEC, we made three changes in decision-making and restart procedures. These changes were meant to make SEC "mimic" short resolution proofs that compute filtering and correlation functions in the topological order of blocks of the specfiication. Of course, since SEC employs conflict driven learning , it deduces clauses that are very different from ones filtering and correlation CNFs consist of. ***The first change*** made in SEC , is that the next branching variable is selected among the topmost clause K of the stack of conflict clauses that has literals of free (i.e. unassigned) variables. That is even if K is satisfied, SEC picks the next branching variable among the free variables of K. Only after all the free variables of K have been assigned, the next topmost clause of the stack having literals of free variables is selected. The effectiveness of this heuristic in decision making can be explained by the necessity to produce clauses relating internal points of both circuits like clauses of a restricting set. Otherwise, if we just *satisfy* the topmost clause of the stack as BerkMin does, there is a danger of getting stuck in one of the two circuits (by producing conflict clauses relating only points of this circuit) or getting biased toward one of the two circuits.

The second change is that among the free variables of K, the variable x with the largest value of $activity(x){*}f(level(x))$ is selected. Here $activity(x)$ is the activity of x computed exactly as in BerkMin, $level(x)$ is the topological level of the gate whose output is specified by x in the miter. The function f is a simple monotonically growing function. The intuition behind this heuristic is that short resolution proofs deduce clauses of restricting and filtering sets in topological order starting from inputs. SAT-solvers like BerkMin (or Chaff) tend to derive clauses in terms of deduced variables. So by giving preference to variables with higher topological levels in decision making we make variables of lower levels more likely to be deduced (and so to be used in conflict clauses.) In other words, this heuristic helps implement the following strategy of conflict clause deduction, given a choice of literals to be used in a conflict clause, try to use literals of variables that are closer to inputs.

The third change is that SEC uses "light" and "heavy" restarts. Before a heavy restart SEC applies the same database cleaning procedure as BerkMin does, while a light restart is not preceded by database cleaning. Light restarts occur very often. Namely, as soon as a conflict occurs at depth greater than 15 of the current search tree and at least one conflict have already occurred before, the current search tree is abandoned. The intuition behind new restart heristic is that each restart can be considered as a way of quickly changing the current set of branching variables. The short resolution proofs described in Section 4 that SEC tried to "simulate" consist of n steps. At each step, filtering functions $Ff(q_1(C))$,$Ff(q_2(C))$ and the correlation function $Cf(q_1(C),q_2(C))$ are computed for a variable C of the specification. Switching to computing these three functions for some other variable C' can be viewed as a restart. This is because sets of variables involved in computing filtering and correlation functions for C and C' are different. So, if the circuits to be checked for equivalence have a fine CS, restarts should be made frequently.

6. Experimental Results

In the experiments we compared the performance of Zchaff (downloaded from [11]), BerkMin (version 561 that can be downloaded from [1]) and SEC. None of the three programs used any kind of formula preprocessing. The experiments were run on a SUNW Ultra-80 system with clock frequency 450MHz. In all the experiments the time limit was set to 60,000 sec. (16.6 hours). In each table the best result is shown in bold.

Table 1. Equivalence checking of MCNC-91 benchmarks

Name	Zchaff sec.	BerkMin sec.	SEC sec.
C3540	125.6	13.7	**5.0**
C5315	75.71	10.5	**3.8**
C6288	60,000	60,000	**211.7**
C880	4.0	0.6	**0.2**
alu4	2.53	**0.9**	1.0
dalu	3.48	**1.6**	2.5
des	367.3	**15.6**	18.4
i8	6.5	**1.8**	3.7
k2	1.7	**0.9**	2.2
t481	8.5	3.8	**3.1**
too_large	265.3	62.2	**48.3**
x1	3.3	**0.5**	1.1

Table 2. Equivalence checking of multipliers

Name	Zchaff sec.	BerkMin sec.	SEC sec.
mlp9	2,287.8	761.5	**10.6**
mlp10	25,858.5	3,854.4	**14.2**
mlp12	60,000	41,315.3	**37.0**
mlp14	60,000	60,000	**51.4**
mlp18	60,000	60,000	**144.6**
mlp24	60,000	60,000	**1,437.8**
mlp32	60,000	60,000	**12,035.1**

First, we tested these programs on "regular" instances where the circuits to be checked for equivalence may have many functionally equivalent points. Table 1 shows the result of equivalence checking for some MCNC-91 circuits. In the experiments we checked for equivalence the original circuit and the circuit obtained by optimization in SIS [10] using a logic optimization script, script.rugged.

Table 3. Equivalence checking of circuits with a CS

Name of topology	Zchaff (sec.)	Berk-Min (sec.)	SEC (sec.)
C880	60,000	200.1	**3.7**
ttt2	799.4	77.2	**11.7**
x4	769.5	139.0	**17.3**
i9	23.5	**11.5**	32.7
term1	60,000	1,183.6	**35.9**
c7552	803.5	74.5	**52.8**
c3540	60,000	4,172.0	**64.1**
rot	60,000	1,346.9	**72.2**
9symml	210.6	**58.2**	113.2
frg2	7,711.0	1,552.9	**131.4**
frg1	60,000	3,602.4	**330.3**
i10	60,000	38,244.6	**445.0**
des	1,390.3	**331.1**	451.7
dalu	60,000	20,234.4	**518.6**
x1	60,000	60,000	**950.2**
alu4	25,503.6	2,372.6	**992.6**
i8	35,721.5	4,039.7	**1,051.5**
c6288	60,000	60,000	**1,955.1**
k2	60,000	60,000	**5,121.5**
too_large	60,000	60,000	60,000
t481	60,000	60,000	60,000

All three programs relatively easily solved all the instances except C6288 (a 16-bit multiplier) that was solved only by SEC. To the best of our knowledge this is the first time when a SAT-solver is able to solve the miter of 16-bit multipliers without deducing internal equivalences or any preprocessing.
Table 2 shows that SEC can handle much larger multipliers (up to a 32-bit multiplier). In the experiments we verified the original circuit of a regular shift-add multiplier against the one obtained by optimization in SIS using script.rugged.

In Table 3 we compare the performance of these three programs on circuits with a CS. These circuits were obtained using the following technique. To get multi-valued specifications with realistic topologies we "borrowed" them from the MCNC-91 benchmark circuits as follows. First all the benchmarks were technology mapped using SIS to get circuits consisting only of two-input AND gates (with possible input inversion). Then from each obtained circuit N a multi-valued specification S was produced by replacing each two-input binary gate with a two-input four-valued gate. (In other words, S changes the functionality of N while preserving its topology.) Then from S two functionally equivalent Boolean circuits N_1, N_2 were produced using two different sets of two-bit encodings of four-valued values. The encodings were picked in such a way that the two different implementations of the same four-valued gate in N_1 and N_2 had no functionally equivalent outputs. That guaranteed that internal functionally equivalent points in N_1 and N_2 may occurr only by accident. Note that after encoding the number of inputs and outputs in N_1 and N_2 is twice the number of inputs and outputs in the original Boolean network N. For instance, the two circuits produced from C6288 have the topology of a 16-bit multiplier and the number of inputs and outputs of a 32-bit multiplier.

It is not hard to see that the performance of BerkMin and Zchaff is much worse on formulas from Table 3 even though they are only a few times larger than formulas of Table 1. SEC does much better than BerkMin or Zchaff but even its performance quickly degrades as the size of the formulas of Table 3 grows. We believe that if we generated circuits from slightly more coarse spcecifications, the performance of SEC (let alone BerkMin and Chaff) would be much worse. On the other hand, an algorithm that knows the CS used when generating the circuits to be checked for equivalence (and that computes filtering and correlation functions as it was described in Section 4) should be able to solve each formula from Table 3 in a few seconds.

7. Conclusions

We introduce a notion of a Common Specification (CS) of Boolean circuits that generalizes the existing notion of structural similarity. We show that the equivalence of circuits N_1,N_2 with a CS S can be proved in general resolution in the number of resolutions that is linear in the number of blocks of S and is exponential in the "size" of the largest block. This suggests that a deterministic algorithm that "knows" S has about the same perfromance (in particular, it is linear in the number of blocks of S). On the other hand, there are good reasons to believe that if S is not known then checking N_1 and N_2 for equivalence is hard. We give some expiremental results that substantiate this claim.

8. References

[1] BerkMin web page. http://eigold.tripod.com/BerkMin.html

[2] Brand, D., *Verification of large synthesized designs*. Proceedings of ICCAD-1993,pp 534-537.

[3] Bryant.,R.E. *Graph based algorithms for Boolean function manipulation.* IEEE Transactions on Computers, C(35):677-691.

[4] Goldberg,E.,and Novikov,Y. *BerkMin: A fast and robust SAT-solver*. Design, Automation, and Test in Europe (DATE '02), pages 142-149, March 2002.

[5] Gupta, A., and Ashar, P. *Integrating Boolean satisfiability checker and BDDs for combinational equivalence checking.* Proc. Int. Conf. on VLSI Design, Chennai, India 1998.

[6] Kuehlmann, A., and Krohm, F. *Equivalence checking using cuts and heaps.* Proceedings of DAC-1997.

[7] Kunz, W., Pradhan, D., *Recursive learning: a new implication technique for efficient solutions to cad problems - test, verification and optimization*. IEEE Tran. on CAD 13(9) Sep. 1994

[8] Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., and Malik, S. *Chaff: Engineering an efficient SAT-solver.* Proceedings of DAC-2001.

[9] Razborov, A., Alekhnovich, M. *Resolution is not automatizable unless W[p] is tractable.* FOCS-2001, pp.210-219.

[10] Sentovich, E. e.a. *Sequential circuit design using synthesis and optimization*. Proceedings of ICCAD, pp 328-333, October 1992.

[11] Zchaff web page. http://ee.princeton.edu/~chaff/zchaff.php