

On Complexity of External and Internal Equivalence Checking

Cadence Berkeley Labs
 1995 University Ave., Suite 460, Berkeley, California, 94704
 phone: (510)-647-2825, fax: (510)-486-0205



CDNL-TR-2006-0105
 January 2006

Eugene Goldberg (Cadence Berkeley Labs), egold@cadence.com
 Kanupria Gulati (Texas A&M University), kanu.gulati@gmail.com

Abstract—We compare the complexity of “internal” and “external” equivalence checking. The former is meant for proving the correctness of a synthesis transformation by which circuit N_2 is obtained from circuit N_1 . The latter is meant for proving that circuits N_1 and N_2 are functionally equivalent without making any explicit assumptions about the origin of N_1 and N_2 . We describe logic synthesis procedures that can produce a circuit N_2 whose equivalence with the original circuit N_1 , most likely, can not be efficiently proved by an external equivalence checker. On the other hand, there are internal equivalence checking procedures that easily prove that N_1 and N_2 are equivalent. We give experimental data showing that these logic synthesis procedures are not a mathematical curiosity but indeed can be used as a powerful method of logic optimization.

Index Terms—internal and external equivalence checking, scalable logic synthesis, toggle equivalence, toggle implication.

I. INTRODUCTION

Equivalence checking has become an important part of design verification [8]. This success can be attributed to a good scalability of the state-of-the-art equivalence checkers. In turn, this scalability is due to two factors. *First*, logic synthesis tools usually do not re-encode state variables and so equivalence checking of two sequential circuits reduces to equivalence checking of combinational circuits bounded by registers and/or primary inputs and outputs. (For this reason, in this report, we consider only equivalence checking of combinational circuits and when we refer to a “circuit” we mean a “combinational circuit”.) *Second*, in many cases circuits can be represented by small BDDs [5]. Then equivalence checking of the corresponding combinational circuits of two designs to be compared can be performed efficiently.

A. Equivalence checking of large circuits

The existence of large combinational circuits poses a problem in equivalence checking. Informally, we consider a circuit as “large”, if its BDD cannot be built efficiently.

Typically, this happens when a circuit has a large width [2]. The width of a circuit N describes the amount of communication between different parts of N , multiplier being a classical example of a “wide” circuit.

A lot of research in equivalence checking has been focused on handling large combinational circuits. In [10], combining SAT and BDD based methods was explored. The case when circuits to be checked for equivalence have compact BDDs under different variable orders was studied in [14]. The most popular method of handling large circuits is based on employing cut points [3][4]. The idea is to prove functional equivalence of circuits N_1 and N_2 inductively. First equivalence of some subcircuits of N_1 and N_2 is established. The outputs of equivalent subcircuits are considered as cut points and new subcircuits are tested for equivalence, inputs of these subcircuits being cut points. This goes on until equivalence of the outputs of N_1 and N_2 is proven in terms of some cut points. This idea was further developed in [11] and successfully used in many equivalence checkers (e.g. [6][7]). It may happen that even though N_1 and N_2 are equivalent, they appear to be inequivalent in terms of the chosen cut points. This situation is usually called a false negative. The problem of false negatives was addressed in [13][15].

B. Internal and external equivalence checking

All the methods described above are meant for “external” equivalence checking. Informally, equivalence checking is external if no explicit assumptions are made about the origin of combinational circuits N_1 and N_2 to be compared. We will say that equivalence checking is “internal” if it is meant for verification of a logic synthesis transformation by which circuit N_2 is obtained from N_1 . So, by definition, internal equivalence checking “knows” the relation between N_1 and N_2 . For simple transformations, internal equivalence checking is performed implicitly by using some informal reasoning. Suppose, for example, that internal points w and w' of a circuit N_1 are functionally equivalent. Then one can optimize N_1 by removing w' and feeding its fan-out nodes with w (instead of

w'). No explicit procedure is run to prove that N_1 and the resulting circuit N_2 are equivalent because in this particular case the equivalence of N_1 and N_2 is “obvious”.

Development of new “non-trivial” methods for internal equivalence checking is extremely important because these methods enable new logic synthesis procedures. A powerful method of internal equivalence checking was introduced in [9] where logic synthesis and equivalence checking of circuits with a **common specification** was considered. A specification of a circuit N is just a partition of N into subcircuits. Two circuits N_1 and N_2 have a common specification if they can be partitioned into k subcircuits N_1^1, \dots, N_1^k and N_2^1, \dots, N_2^k such that these subcircuits are connected in “the same way” in N_1 and N_2 and corresponding subcircuits N_1^i, N_2^i are toggle equivalent. We will refer to the equivalence checking procedure of [9] as **EC_TE** where EC stands for equivalence checking and TE stands for toggle equivalence.

The importance of EC_TE is twofold. *First*, it enables a powerful logic synthesis procedure. (We will refer to this procedure as **LS_TE** where LS stands for logic synthesis and TE for toggle equivalence). Given a circuit N_1 with specification N_1^1, \dots, N_1^k , LS_TE builds an optimized circuit N_2 that is functionally equivalent to N_1 by replacing subcircuits N_1^i with their toggle equivalent counterparts in topological order. (The equivalence of N_1 and N_2 can be established by EC_TE). The power of LS_TE is in its good scalability and flexibility. LS_TE has linear complexity in the number k of subcircuits N_1^i and exponential in the **granularity of specification** which is the size of the largest subcircuit N_1^i measured in the number of gates. (The complexity of EC_TE is the same as that of LS_TE.) If a subcircuit N_1^i of N_1 has m outputs, then the number of m -output subcircuits that are toggle equivalent to N_1^i is huge even for small values m . So LS_TE enjoys great flexibility even for specifications of very small granularity.

The *second* reason why introduction of EC_TE is important is as follows. Usually, external equivalence checking is assumed to be as powerful as internal. The results of [9] imply, that, most probably, there is an exponential gap between external and internal equivalence checking. The reason is that finding a common specification of N_1 and N_2 most likely can not be done efficiently. So equivalence checking of N_1 and N_2 becomes infeasible if the partitioning of N_1 and N_2 into subcircuits N_1^1, \dots, N_1^k and N_2^1, \dots, N_2^k is not known. In other words, if N_1 is a circuit of large width, external equivalence checking of N_1 and N_2 obtained from N_1 by a logic optimization procedure is possible only if this procedure is very “weak”. Then N_1 and N_2 have so much similarity (like existence of many functionally equivalent internal points) that equivalence of N_1 and N_2 can be proven without any additional information.

C. Our contribution and structure of the report

In this report, we further develop the ideas of [9]. Our contribution is threefold. In [9], no concrete procedure that, given a subcircuit N_1^i , generates a toggle equivalent

counterpart N_2^i was introduced. So one could consider the predicted performance gap between internal and external equivalence checking as a mathematical curiosity. The *first* contribution of this report is that we give experimental data showing that LS_TE has great potential, which makes the theory of [9] much more tangible.

The *second* contribution is that we show that after a slight modification, EC_TE enables a logic synthesis procedure much more powerful than LS_TE. Given a circuit N_1 and specification N_1^1, \dots, N_1^k , this procedure is to replace each subcircuit N_1^i with a subcircuit N_2^i that implies toggling of N_1^i . We will refer to this procedure as **LS_TI** (where TI stands for toggle implication). LS_TI offers much more flexibility than LS_TE while its complexity is the same as the complexity of LS_TE. The *third* contribution of this report is that we show that LS_TI “widens” the gap between internal and external equivalence checking.

This report is structured as follows. In Section II we give basic notions. Section III recalls EC_TE and LS_TE procedures of [9]. In Section IV we describe an equivalence checking procedure that enables a logic synthesis procedure LS_TI more powerful than LS_TE. Section V gives experimental data showing the power of LS_TE and explains why LS_TI is more powerful than LS_TE. In Section VI we discuss the complexity of external equivalence checking of circuits produced by LS_TE and LS_TI. We conclude with Section VII.

II. BASIC NOTIONS

A. Toggle equivalence of Boolean functions

In this subsection, we recall the notion of toggle equivalence and its properties. All the propositions given in this section are either proven in [9] or can be easily derived from proofs given there.

Definition 1. Let $f: \{0,1\}^n \rightarrow \{0,1\}^m$ be an m -output Boolean function. A **toggle** of f is a pair of two different output vectors produced by f for two input vectors. In other words, if $y=f(x)$ and $y'=f(x')$ and $y \neq y'$, then (y, y') is a toggle.

Definition 2. Let f_1 and f_2 be m -output and k -output Boolean functions of the same set of variables. Functions f_1 and f_2 are called **toggle equivalent** if $f_1(x) \neq f_1(x') \Leftrightarrow f_2(x) \neq f_2(x')$. Circuits N_1 and N_2 implementing toggle equivalent functions f_1 and f_2 are called **toggle equivalent circuits**.

Proposition 1. Let $f_1: \{0,1\}^n \rightarrow \{0,1\}^m$ and $f_2: \{0,1\}^n \rightarrow \{0,1\}^k$ be m -output and k -output Boolean functions of the same set of variables. Let f_1 and f_2 be toggle equivalent. Then there is an invertible function K such that $f_1(x)=K(f_2(x))$ and $f_2(x)=K^{-1}(f_1(x))$.

Proposition 1 means that if functions f_1 and f_2 are toggle equivalent, then there is a one-to-one mapping K between the output vectors produced by f_1 and f_2 .

Proposition 2. Let f_1 and f_2 be toggle equivalent single output Boolean functions. Then $f_1=f_2$ or $f_1=\sim f_2$ where ‘ \sim ’ means negation.

Let N_1 and N_2 be toggle equivalent functions. Definition 3, Definition 4 and Proposition 3 below explain how one can *implicitly* find the mapping K relating outputs produced by N_1 and N_2 .

Definition 3. Let f be a Boolean function. We will say that function f^* is obtained from f by *existentially quantifying away* variable x_i if $f^* = f(\dots, x_i=0, \dots) \vee f(\dots, x_i=1, \dots)$.

Definition 4. Let N be a circuit. Denote by $v(N)$ the set of variables of N . Denote by $Sat(v(N))$ the Boolean function such that $Sat(z)=1$ iff the assignment z to $v(N)$ is “possible” i.e consistent. For example, if N consists of just one AND gate $y=x_1 \wedge x_2$, then $Sat(v(N)) = (\sim x_1 \vee \sim x_2 \vee y) \wedge (x_1 \vee \sim y) \wedge (x_2 \vee \sim y)$. For the sake of simplicity we will denote $Sat(v(N))$ as $Sat(N)$.

Proposition 3. Let N_1 and N_2 be toggle equivalent and Y_1, Y_2 be the sets of their output variables. Let function $K^*(Y_1, Y_2)$ be obtained from $Sat(N_1) \wedge Sat(N_2)$ by existentially quantifying away the variables of N_1 and N_2 except those of $Y_1 \cup Y_2$. The function $K^*(Y_1, Y_2)$ implicitly specifies the one-to-one mapping K between output vectors produced by N_1 and N_2 . Namely $K^*(y_1, y_2)$ is equal to 1 iff $y_1=K(y_2)$.

B. Implication of toggling

In this subsection, we introduce the notion of implication of toggling.

Definition 5. Let f_1 and f_2 be two multi-output functions with the same set of variables $X=\{x_1, \dots, x_n\}$. Toggling of function f_1 **implies toggling of** f_2 , if for any pair of assignments x', x'' to the variables of X , $f_1(x') \neq f_1(x'')$ implies $f_2(x') \neq f_2(x'')$.

Definition 6. Toggling of a multi-output function $f_1(x_1, \dots, x_n)$ **strictly implies toggling of** a multi-output function $f_2(x_1, \dots, x_n)$ if toggling of f_1 implies toggling of f_2 and there is a pair of assignments x', x'' to the variables of X such that $f_1(x') = f_1(x'')$ while $f_2(x') \neq f_2(x'')$.

Remark 1. We will **denote** by $f_1 \leq f_2$ (respectively $f_1 < f_2$) the fact that toggling of function f_1 implies toggling of (respectively strictly implies toggling of) f_2 . We will **denote** by $N_1 \leq N_2$ (respectively $N_1 < N_2$) the fact that toggling of the function implemented by Boolean circuit N_1 implies toggling of (respectively strictly implies toggling of) the function implemented by Boolean circuit N_2 .

Proposition 4. Boolean functions f_1 and f_2 are toggle equivalent iff $f_1 \leq f_2$ and $f_2 \leq f_1$.

Proposition 5. Let $f_1: \{0,1\}^n \rightarrow \{0,1\}^m$ and $f_2: \{0,1\}^n \rightarrow \{0,1\}^k$ be m -output and k -output Boolean functions of the same set of variables. Let $f_1 \leq f_2$. Then there is a function K such that $f_1(x)=K(f_2(x))$.

Note that unless f_1 and f_2 are toggle equivalent, the function K is not invertible.

C. Testing toggle implication and toggle equivalence

In this subsection, we describe how toggle equivalence and implication of toggling can be tested

Let N_1 and N_2 be two Boolean circuits to be checked for implication of toggling. Let $X=\{x_1, \dots, x_n\}$ be the set of input

variables of N_1, N_2 . Let $Y=\{y_1, \dots, y_m\}$ and $Z=\{z_1, \dots, z_k\}$ be the sets of output variables of N_1 and N_2 respectively. Then $N_1 \leq N_2$ holds iff the function $S = H(N_1, N_2) \wedge H(N_1^*, N_2^*) \wedge Neg(Y, Y^*) \wedge Eq(Z, Z^*)$ is unsatisfiable (i.e. it is a constant 0). Here N_1^* and N_2^* are copies of circuits N_1 and N_2 , with input variables $X^*=\{x_1^*, \dots, x_n^*\}$ and output variables $Y^*=\{y_1^*, \dots, y_m^*\}$ and $Z^*=\{z_1^*, \dots, z_k^*\}$ respectively. The function $H(N_1, N_2)$ is equal to $Sat(N_1) \wedge Sat(N_2)$. The value of $Eq(y, y^*)$ where y and y^* are assignments to Y and Y^* respectively is equal to 1 iff $y=y^*$. The function $Neg(Y, Y^*)$ is the negation of $Eq(Y, Y^*)$.

Indeed, $S=1$ means that for a pair of input vectors x and x^* , circuit N_1 toggles (which sets $Neg(Y, Y^*)$ to 1) while N_2 does not (which sets $Eq(Z, Z^*)$ to 1).

From Proposition 4 it follows that checking for toggle equivalence reduces to two satisfiability checks (SAT-checks for short).

D. Correlation function

In this subsection, we use the notion of correlation function to extend definitions of toggle implication and toggle equivalence to the case when functions f_1 and f_2 have different sets of variables.

Definition 7. Let X and X^* be two disjoint sets of Boolean variables (the number of variables in X and X^* may be different). A function $D(X, X^*)$ is called a **correlation function** if there are subsets $Q^X \subseteq \{0,1\}^{|X|}$ and $Q^{X^*} \subseteq \{0,1\}^{|X^*|}$ such that $D(X, X^*)$ specifies a bijective mapping $M: Q^X \rightarrow Q^{X^*}$. Namely $D(x, y)=1$ iff $x \in Q^X$ and $y \in Q^{X^*}$ and $y = M(x)$.

Let $f_1(X)$ and $f_2(X^*)$ be two multi-output Boolean functions where $X=\{x_1, \dots, x_k\}$ and $X^*=\{x_1^*, \dots, x_p^*\}$ are sets of their variables. (Note, that f_1 and f_2 may have different number of variables.). Let $D(X, X^*)$ be a correlation function relating variables of f_1 and f_2 . Then one can introduce notions of toggle equivalence and toggle implication for f_1 and f_2 . The only difference from definitions and results listed in subsections A,B,C is that now one should consider only assignments that satisfy $D(X, X^*)$.

Let us show how this works for toggle equivalence. Functions $f_1(X)$ and $f_2(X^*)$ are said to be toggle equivalent under input constraint $D(X, X^*)$, if for any two pairs (x, x^*) and (y, y^*) of input vectors such that $D(x, x^*)=D(y, y^*)=1$, it is true that $f_1(x) \neq f_1(y) \Leftrightarrow f_2(x^*) \neq f_2(y^*)$. The mapping between output vectors produced by toggle equivalent circuits N_1 and N_2 (implementing functions f_1 and f_2 respectively), can be obtained from $Sat(N_1) \wedge Sat(N_2) \wedge D(X, X^*)$ by existentially quantifying away all the variables of N_1 and N_2 except output variables. The other results and definitions of subsections A,B,C can be modified in a similar manner.

III. LOGIC SYNTHESIS AND EQUIVALENCE CHECKING OF CIRCUITS WITH COMMON SPECIFICATION

In this section, we recall LS_TE and EC_TE procedures of [9]. From now on we assume that circuit N_1 to be optimized has only one output. (If a circuit to be optimized has more

than one output, then the LS_TE procedure can be separately applied to every subcircuit feeding an output of N_1).

A. Logic synthesis preserving toggle equivalence

In this subsection, we recall the procedure of Logic Synthesis preserving Toggle Equivalence (abbreviated as **LS_TE**) introduced in [9]. The pseudocode of the LS_TE procedure is shown in Figure 1.

```

1 LS_TE( $N_1, Spec(N_1), cost\_function$ ) {
2   for ( $i=1; i \leq k; i++$ ) {
3      $D_{inp}(N_1^i, N_2^i) = constraint\_function(N_1, N_2, i);$ 
4      $N_2^i = synth\_toggle\_equivalent(N_1^i, D_{inp}, cost\_function)$ 
5      $D_{out}(N_1^i, N_2^i) = exist\_quantify(N_1^i, N_2^i, D_{inp});$  }
6   return( $N_2, Spec(N_2)$ )}

```

Figure 1. Pseudocode of the LS_TE procedure

Following [9] we also assume that specification $Spec(N_1) = \{N_1^1, \dots, N_1^k\}$ (i.e. the initial partition of circuit N_1 into subcircuits N_1^i) is topological. Let G be a directed graph whose nodes are subcircuits N_1^i and an edge of G directed from node N_1^i to node N_1^j implies that an output of N_1^i is connected to an input of N_1^j . We will call G a **specification graph**. $Spec(N_1)$ is a **topological specification** if its specification graph G is acyclic.

Given a circuit N_1 with specification $Spec(N_1) = \{N_1^1, \dots, N_1^k\}$, LS_TE builds circuit N_2 with specification $Spec(N_2) = \{N_2^1, \dots, N_2^k\}$ that is identical to $Spec(N_1)$. Let G_1 and G_2 be directed graphs describing connections of subcircuits N_1 and N_2 as described above. $Spec(N_1)$ and $Spec(N_2)$ are considered to be **identical specifications** if a) G_1 and G_2 are acyclic; b) G_1 and G_2 are isomorphic; c) subcircuits N_1^i and N_2^i are toggle equivalent, $i=1, \dots, k$ in terms of their inputs related by a constraint function that is a correlation function (see below).

Since $Spec(N_1)$ is topological, one can assign levels to subcircuits N_1^i . We assume that subcircuits N_1^i are numbered ‘‘topologically’’ i.e. if $i < j$ then $topol_level(N_1^i) \leq topol_level(N_1^j)$. The LS_TE procedure builds circuit N_2 by replacing subcircuits N_1^i , $i=1, \dots, k$ with their toggle equivalent counterparts N_2^i in topological order, from inputs to outputs.

Let us consider how LS_TE works by the example of Figure 2. LS_TE starts with subcircuit N_1^1 and recovers the function $D_{inp}(N_1^1, N_2^1)$ relating inputs of N_1^1 and subcircuit N_2^1 to be built (line 3 of the pseudocode). The inputs of N_1^1 are inputs of N_1 (and so N_1^1 has the lowest topological level 1). In that case, $D_{inp}(N_1^1, N_2^1)$ is just a conjunction of equality functions relating corresponding inputs of N_1^1 and N_2^1 (and so $D_{inp}(N_1^1, N_2^1)$ is a correlation function ‘‘identifying’’ the corresponding inputs of N_1^1 and N_2^1 .) Then an actual subcircuit N_2^1 toggle equivalent to N_1^1 is synthesized (line 4). In the end of this iteration, the function $D_{out}(N_1^1, N_2^1)$ relating outputs of N_1^1 and N_2^1 is built (line 5) as described in Proposition 3. Since N_1^1 and N_2^1 are toggle equivalent, there is a one-to-one mapping between the output vectors they produce. So $D_{out}(N_1^1, N_2^1)$ is a correlation function.

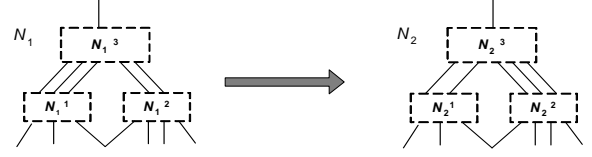


Figure 2. Optimization of N_1 by the LS_TE procedure

Then, the LS_TE procedure processes subcircuit N_1^2 in the same manner, generating a toggle equivalent subcircuit N_2^2 and the correlation function $D_{out}(N_1^2, N_2^2)$. Finally, the subcircuit N_1^3 is processed similarly to N_1^1 and N_1^2 with one exception. The inputs of N_1^3 are fed by the outputs of N_1^1 and N_1^2 . So now the function $D_{inp}(N_1^3, N_2^3)$ relating inputs of N_1^3 and N_2^3 (synthesized in line 4) equals $D_{out}(N_1^1, N_2^1) \wedge D_{out}(N_1^2, N_2^2)$. (To obtain $D_{inp}(N_1^3, N_2^3)$ one has to take the conjunction of $D_{out}(N_1^i, N_2^i)$ for all the subcircuits whose outputs feed inputs of N_1^3 and N_2^3 . In this particular case, these subcircuits are N_1^1, N_1^2 and N_2^1, N_2^2 .) It is not hard to show that a conjunction of correlation functions is a correlation function too and so $D_{inp}(N_1^3, N_2^3)$ is a correlation function.

Let us N_2^3 be a circuit toggle equivalent to N_1^3 under constraints $D_{inp}(N_1^3, N_2^3)$. Recall that N_1^3 has only one output. It is not hard to show that then N_2^3 either has one output or all outputs of N_2^3 but one can be removed without affecting its being toggle equivalent to N_1^3 . (So we will assume that N_2^3 has one output.) Since N_1^3 and N_2^3 single-output subcircuits, their toggle equivalence, means that they are functionally equivalent modulo negation. So the circuit N_2 consisting of subcircuits N_2^1, N_2^2, N_2^3 is functionally equivalent to N_1 (modulo negation).

B. Equivalence checking of circuits with a common specification

The equivalence checking of N_1 and a circuit N_2 obtained from N_1 by LS_TE can be done by the **EC_TE** procedure of [9] whose (slightly changed) pseudocode is shown in Figure 3. EC stands for equivalence checking and TE stands for toggle equivalence. Recall that N_1 is a single-output circuit and, as we mentioned above, LS_TE builds a circuit N_2 that has one output too.

The input to the EC_TE procedure are circuits N_1 and N_2 and their specifications $Spec(N_1) = \{N_1^1, \dots, N_1^k\}$, $Spec(N_2) = \{N_2^1, \dots, N_2^k\}$. So EC_TE is an internal equivalence checking procedure. EC_TE is incomplete in the sense that it gives a definite answer only if $Spec(N_1)$ and $Spec(N_2)$ are identical. Otherwise, EC_TE returns the answer ‘*CS_check_failure*’, which means that specifications of N_1 and N_2 are different.

Let G_1, G_2 be specification graphs for $Spec(N_1)$ and $Spec(N_2)$ respectively (see subsection A.) In line 2, EC_TE checks if graphs G_1 and G_2 are acyclic. In line 4, EC_TE checks if G_1 and G_2 are isomorphic. If either check fails, ‘*CS_check_failure*’ is reported.

```

1 EC_TE( $N_1, N_2, Spec(N_1), Spec(N_2)$ ) {
2 if ( $topol\_spec(N_1, Spec(N_1)) == 'no'$ ) ||  $topol\_spec(N_2, Spec(N_2)) == 'no'$ )
3   return('CS_check_failure');
4 if ( $graph\_isomorphism(N_1, N_2, Spec(N_1), Spec(N_2)) == 'no'$ )
5   return('CS_check_failure');
6 for ( $i=1; i \leq k; i++$ ) {
7    $D_{imp}(N_1^i, N_2^i) = constraint\_function(N_1, N_2, i)$ ;
8   if ( $toggle\_equiv((N_1^i, N_2^i, D_{imp})) == 'no'$ ) return('CS_check_failure');
9    $D_{out}(N_1^i, N_2^i) = exist\_quantify(N_1^i, N_2^i, D_{imp})$ ;}
10 if ( $D_{out}(N_1^k, N_2^k)$  is 'equivalence_function') return('equivalent');
11 else return('inequivalent');}

```

Figure 3. Pseudocode of the EC_TE procedure

The main work is done in the ‘for’ loop (lines 6-9) where subcircuits N_1^i and N_2^i ($i=1, \dots, k$) are checked for toggle equivalence in topological order. First, in line 7 the function $D_{imp}(N_1^i, N_2^i)$ relating inputs of N_1^i and N_2^i is formed exactly as it is done by LS_TE (Figure 1, line 3). In line 8, EC_TE checks if N_1^i and N_2^i (whose inputs are related by the function $D_{imp}(N_1^i, N_2^i)$ computed in line 7) are toggle equivalent. If not, then EC_TE returns ‘CS_check_failure’. (As we showed in subsection II.C, checking toggle equivalence reduces to two SAT-checks.) In line 9, the function $D_{out}(N_1^i, N_2^i)$ relating outputs of N_1^i and N_2^i is computed. This is done by existentially quantifying away from $Sat(N_1^i) \wedge Sat(N_2^i) \wedge D_{imp}(N_1^i, N_2^i)$ all the variables except the output variables of N_1^i and N_2^i .

After finishing the ‘for’ loop, in lines 10-11, EC_TE checks if the function $D_{out}(N_1^k, N_2^k)$ (note that the outputs of N_1^k and N_2^k are outputs of N_1 and N_2 respectively) is an equivalence function. If it is, then N_1 and N_2 are equivalent. Otherwise, N_1 and N_2 are complements of each other (because N_1^k and N_2^k are toggle equivalent but not functionally equivalent) and so they are inequivalent.

The complexity of both EC_TE and LS_TE is linear in the number of subcircuits in N_1 and N_2 and exponential in granularities of $Spec(N_1)$ and $Spec(N_2)$. (Recall that **granularity** of $Spec(N_1)$ is the number of gates in the largest subcircuit $N_1^i, i=1, \dots, k$.)

IV. NEW EQUIVALENCE CHECKING AND LOGIC SYNTHESIS PROCEDURES

In this section, we describe an extension of the EC_TE and LS_TE procedures called **EC_TI** and **LS_TI** respectively (where TI stands for toggle implication). First, we give a generic method for introducing a logic transformation through an “enabling” equivalence checking procedure and explain how this method works for EC_TE and LS_TE. Then we introduce EC_TI and LS_TI.

A. A generic method for introducing logic transformation

The idea of the method is to introduce a logic transformation through an Enabling internal Equivalence Checking procedure (we will refer to it as **EEC**). The input to an EEC is an original circuit N_1 , a modified circuit N_2 and some information about the transformation T used to obtain N_2 from N_1 . EEC has to be sound. That is, if EEC says that N_1 and N_2 are equivalent (or inequivalent) it has to give the right

answer. Besides, EEC should be able to recognize if N_2 can not be obtained from N_1 by the transformation T . After designing an EEC, one formulates a logic synthesis procedure that given a circuit N_1 generates a circuit N_2 whose equivalence to N_1 can be verified by this EEC. We will say that this synthesis procedure is enabled by this EEC.

Let us illustrate how this method works for introducing LS_TE. The EC_TE procedure satisfies the requirements above for an EEC. Indeed, the input to EC_TE consists of circuits N_1 and N_2 and partitions $Spec(N_1)$, $Spec(N_2)$ as information about the synthesis transformation to be enabled. The soundness of EC_TE trivially follows from the fact that EC_TE returns the ‘equivalent’ (or ‘inequivalent’) answer only if it correctly derived the equivalence (respectively inequivalence) function relating the outputs of N_1 and N_2 . It is not hard to see that LS_TE is exactly the procedure enabled by EC_TE. Indeed, EC_TE checks in topological order if subcircuits N_1^i and N_2^i of $Spec(N_1)$ and $Spec(N_2)$ are toggle equivalent. In turn, LS_TE builds N_2 by generating toggle equivalent counterparts of subcircuits N_1^i in topological order.

B. Introduction of LS_TI

Let EC_TI be an equivalence checking procedure that is different from EC_TE only in one aspect. Instead of checking if N_1^i and N_2^i (line 8 of Figure 3) are toggle equivalent it checks if toggling of N_1^i implies that of $N_2^i, i=1, \dots, k-1$. For N_1^k and N_2^k it checks toggle equivalence as before. That is for all pairs of subcircuits except for N_1^k, N_2^k , instead of checking if both $N_1^i \leq N_2^i$ and $N_2^i \leq N_1^i$ hold, EC_TI just checks if $N_1^i \leq N_2^i$. The EC_TI procedure is sound for the same reason as EC_TE. (That is it correctly derives an (in)equivalence function relating the outputs of N_1 and N_2 . The correctness of derivation follows from the “soundness” of existential quantification that is used to obtain functions $D_{out}(N_1^i, N_2^i) i=1, \dots, k$)

In the context of the EC_TI procedure, we will say that circuits N_1 and N_2 have **identical specifications** $Spec(N_1) = \{N_1^1, \dots, N_1^k\}$ and $Spec(N_2) = \{N_2^1, \dots, N_2^k\}$ if specification graphs G_1 and G_2 are acyclic, isomorphic and $N_1^i \leq N_2^i, i=1, \dots, k-1$ and for the last value of i (i.e. $i=k$), subcircuits N_1^k and N_2^k are toggle equivalent. The EC_TI procedure either correctly identifies circuits N_1 and N_2 as (in)equivalent or reports that $Spec(N_1)$ and $Spec(N_2)$ are not identical.

Now we define a procedure enabled by EC_TI. Let N_1 be a circuit and $Spec(N_1) = \{N_1^1, \dots, N_1^k\}$ be its specification. Let LS_TI be a logic synthesis procedure that is different from LS_TE only in two aspects. *First*, for $i=1, \dots, k-1$ LS_TI generates subcircuit N_2^i such that toggling of N_1^i implies that of N_2^i i.e. $N_1^i \leq N_2^i$ (So, LS_TI is different from LS_TE in line 4 of Figure 1). Only for $i=k$, LS_TI generates subcircuit N_2^k that is toggle equivalent to N_1^k .

Second, when generating a subcircuit N_2^i , LS_TI limits the number of outputs of N_2^i . The reason is as follows. Let $D_{imp}(N_1^i, N_2^i)$ be the constraint function relating inputs of N_1^i and the subcircuit N_2^i to be built. If the number of outputs in N_2^i is not limited, then the simplest subcircuit N_2^i such that $N_1^i \leq N_2^i$ is the **identity circuit** I_p where p is the number of inputs

in N_2^i . (I_p is a circuit with inputs x_1, \dots, x_p and p outputs y_1, \dots, y_p , implementing the functions $y_i = x_i$, $i=1, \dots, p$. So I_p does not have any gates.) Hence, without limiting the number of outputs in N_2^i , LS_TI would just generate identity circuits as N_2^i , $i=1, \dots, k-1$ pumping all the functionality into functions $D_{\text{inp}}(N_1^i, N_2^i)$. Only when generating the subcircuit N_2^k (which is toggle equivalent to N_1^k and has one output) a non-empty subcircuit would be generated. It can be shown that in this case $D_{\text{inp}}(N_1^k, N_2^k)$, would essentially describe the relation between inputs of N_1^k and the primary inputs of N_1 . This means that if one does not limit the number of outputs in N_2^i , the LS_TI procedure would delay all the synthesis work until $i=k$ and so it would not be scalable. The simplest way to limit the number of outputs in N_2^i is to require that $\text{num_of_outputs}(N_2^i) \leq \text{num_of_outputs}(N_1^i)$. Under such a restriction, LS_TI has the same complexity as LS_TE i.e. it is linear in the number of subcircuits in $\text{Spec}(N_1)$ and exponential in the granularity of $\text{Spec}(N_1)$.

LS_TI is a synthesis procedure enabled by EC_TI. That is, if for a given N_1 and specification $\text{Spec}(N_1)$, LS_TI builds a circuit N_2 with specification $\text{Spec}(N_2)$, EC_TI will prove N_1 and N_2 to be equivalent.

V. BIG PROMISE OF LOGIC SYNTHESIS PRESERVING COMMON SPECIFICATION

In this section we give some experimental data showing the power of the LS_TE procedure and discuss the potential of LS_TI that should be much more powerful than LS_TE.

A. Power of LS_TE

The key procedure of LS_TE is called in the loop (line 4 of Figure 1) k times to generate a subcircuit N_2^i that is toggle equivalent to N_1^i , $i=1, \dots, k$. We will refer to it as the **TEP** procedure (TEP stands for **Toggle Equivalence Preserving**). Unfortunately, no concrete TEP procedure was described in [9]. Such a procedure has been developed in [18]. (For the lack of space, we omit its description.) In this report we give some experimental data on using the TEP procedure for optimization of multi-output circuits just to show that LS_TE has a great practical potential. So the fact that external equivalence checking of circuits produced by LS_TE is problematic is significant.

In Table 1 we compare the results of optimization of some MCNC benchmarks by SIS [16] and by the TEP procedure. The name of the circuit and the number of inputs and outputs are shown in the first three columns of Table 1. The results of optimization by SIS with the script ‘rugged’ followed by technology decomposition (to obtain a circuit of two-input AND gates and invertors) is shown in the fifth column. The results of using TEP to build a toggle equivalent circuit are shown in the fourth (the number of outputs) and sixth (the number of gates) columns.

For the majority of circuits TEP was able to find much smaller toggle equivalent counterparts. In two cases (5xp1 and f51m), TEP removed all the logic. This means that, for

example, for different input assignments, circuit 5xp1 generates different output assignments. So the identity circuit I_7 is toggle equivalent to 5xp1.

Table 1. Generation of toggle equivalent circuits

Circuit	#in-puts	Initial #outputs	TEP #outputs	SIS #gates	TEP #gates
squar5	5	8	8	60	4
rd84	8	4	8	174	52
5xp1	7	10	7	140	0
b1	3	4	3	11	2
bw	5	28	8	155	9
cm138	6	8	4	28	15
cm42a	4	10	6	31	6
cm82a	5	3	5	21	18
exp5p	8	63	19	286	131
f51m	8	8	8	101	0
con1	7	4	8	82	94
sqrt	8	4	9	76	90

Of course, such re-encoding of output assignments of the original circuit requires changing the surrounding logic. To explain why re-encoding may still lead to significant logic reduction let us consider the following example. Suppose that a circuit N_1 to be optimized consists of two subcircuits, N_1^1 and N_1^2 where the outputs of N_1^1 are connected to inputs of N_1^2 . Suppose circuits N_1^1 and N_1^2 were built “independently”. That is when designing circuit N_1^1 , the output encoding for N_1^1 was chosen without any consideration of circuit N_1^2 . Then, in a sense, any circuit toggle equivalent to N_1^1 is as good as N_1^1 . So, it is a reasonable heuristic to build a circuit N_2^1 that is the smallest toggle equivalent counterpart of N_1^1 and then try to find the smallest subcircuit N_2^2 that is toggle equivalent to N_1^2 under constraints specified by the function $D_{\text{out}}(N_1^1, N_1^2)$.

Table 2 shows results of applying LS_TE with the heuristic above to logic optimization of some arithmetic expressions with an integer variable x . (The second column of Table 2 gives the number of bits in x .) Each circuit N_1 of Table 2 consists of subcircuits N_1^1 and N_1^2 . First three circuits N_1 implement Boolean function $x^2 < C_1$. Here N_1^1 implements the function x^2 and subcircuit N_1^2 implements comparison with a constant C_1 . The last three circuits N_1 implement Boolean function $C_1 * x < C_2$. Here, subcircuit N_1^1 implements the function $C_1 * x$, and N_1^2 implements comparison with a constant C_2 . Circuits N_1^1 and N_1^2 were built from standard combinational blocks. (So, for example, in our experiments, N_1^1 was a derivative of a regular multiplier.)

Table 2. Optimization by LS_TE

Circuit	#bits	C_1	C_2	SIS #gates	LS_TE #gates
$x^2 < C_1$	6	200	-	196(12)	5
$x^2 < C_1$	7	200	-	265 (16)	6
$x^2 < C_1$	7	500	-	273(15)	6
$C_1 * x < C_2$	7	49	300	84(15)	6
$C_1 * x < C_2$	7	111	300	160(12)	6
$C_1 * x < C_2$	7	49	500	56(14)	6

It is not hard to see that $x^2 < C_1$ is equivalent to $x < C'_1$ where C'_1 is the constant equal to $\text{ceiling}(\text{square_root}(C_1))$.

Similarly, $C_1 * x < C_2$ is equivalent to $x < C'_2$ where $C'_2 = \text{ceiling}(C_2/C_1)$. So there is a very simple circuit implementation of either Boolean function.

The results of optimization by SIS are shown in the fifth column. The first number of this column gives the number of gates in the circuit obtained after applying script ‘rugged’ and technology decomposition. The number in parenthesis gives the number of gates in the resulting circuit after applying script ‘rugged’ many times until the solution stabilizes and then running technology decomposition.

The results of applying LS_TE (that used the TEP procedure of [18]) are shown in the last column. Note that N_1^1 (for both x^2 and $C_1 * x$ cases) generates different output assignments for different input assignments. This means that the identity circuit I_m (where m is the number of bits in x) is toggle equivalent to N_1^1 . So LS_TE picked I_m as N_2^1 . Then it computed the function $D_{\text{out}}(N_1^1, N_2^1)$ relating outputs of N_1^1 and N_2^1 and built subcircuit N_2^2 toggle equivalent to N_1^1 under constraints specified by $D_{\text{out}}(N_1^1, N_2^1)$. The size of circuits obtained by LS_TE is smaller than those of SIS even after multiple applications of the script ‘rugged’. (Since a circuit built by LS_TE is functionally equivalent to the original one, it is a fair comparison.)

Note, that indeed LS_TE applied the heuristic mentioned above. Since N_1^1 implementing, say, x^2 was built without any consideration of N_2^1 any circuit toggle equivalent to N_1^1 is as “good” as N_1^1 . So LS_TE picked the simplest such a circuit that is I_m . Interestingly, by replacing N_1^1 implementing x^2 with I_m (implementing just function x), LS_TE essentially “takes the square root” of x^2 . In turn, subcircuit N_2^2 built by LS_TE implements the expression $x < \text{ceiling}(\text{square_root}(C_1))$. So LS_TE is able to perform “intelligent” optimization at the gate level. In particular, it can make high-level algebraic transformations “implicitly” using re-encoding.

B. Potential of LS_TI

LS_TI is more powerful than LS_TE because toggle implication is a more general relation than toggle equivalence. So LS_TI is much more flexible than LS_TE (while its complexity is the same as that of LS_TE).

Let N_1^i be a subcircuit of circuit N_1 with specification $\text{Spec}(N_1) = \{N_1^1, \dots, N_1^k\}$. If N_2^i is a subcircuit synthesized by LS_TE that is toggle equivalent to N_1^i (under constraints specified by function $D_{\text{inp}}(N_1^i, N_2^i)$), every toggle of N_2^i has a “matching” toggle of N_1^i . On the other hand, if N_2^i is built by LS_TI, it may have toggles that are not matched by N_1^i (because LS_TI has to preserve only $N_1^i \leq N_2^i$). Since LS_TI builds a circuit N_2 that is functionally equivalent to N_1 , these unmatched toggles of N_2^i do not reach the output of N_2 . The blocking of the unmatched toggles is done “automatically”.

This flexibility of LS_TI can be viewed as a kind of unobservability Don’t-Cares. The difference from “regular” unobservability Don’t-Cares is twofold. *First*, such Don’t-Cares are usually computed with respect to one wire. *Second*, using unobservability Don’t-Cares should preserve the functionality of the entire circuit. This makes their computation unscalable. On the other hand, in LS_TI, these

“Don’t Cares” are inherently local. At the time of building N_2^i , the part of the final circuit N_2 between outputs of N_2^i and the output of N_2 is not even known yet.

Let us consider advantage of LS_TI over LS_TE by a simple example. Suppose that one needs to implement Boolean function $f(x) < 9$ where x is an m -bit integer. Let $f(x)$ be equal to x^2 at all the 2^m points except for the point $x=4$ where $f(4)$ is equal to 25 (instead of 16). It is not hard to see that the expression $f(x) < 9$ is equivalent to $x < 3$. Suppose that $f(x) < 9$ is implemented by a circuit N_1 that is a composition of subcircuits N_1^1 and N_1^2 where N_1^1 implements $f(x)$ and N_1^2 implements comparison with 9.

Suppose that the LS_TE procedure is applied to optimize N_1 . LS_TE can not use I_m as N_2^1 , because $f(4) \neq f(5) = 25$ and so I_m is not toggle equivalent to N_1^1 . So LS_TE would build N_2^1 implementing some “non-trivial” function of x . Hence, N_2^2 would have to implement a more complex function than $x < 3$.

Now suppose that N_1 is optimized by LS_TI. Note that toggling of N_1^1 implies toggling of the identity circuit I_m . So LS_TI can use I_m as subcircuit N_2^1 . Then LS_TI can build subcircuit N_2^2 implementing $x < 3$. So due to greater flexibility LS_TI is able to generate a smaller circuit than LS_TE.

To become practical, LS_TI needs a procedure that, given a subcircuit N_1^i and a constraint function D_{inp} , builds an optimized subcircuit N_2^i such that a) $N_1^i \leq N_2^i$ under constraint D_{inp} ; b) the number of outputs in N_2^i is limited by a constant. (We will refer to this procedure as **Toggle Implication Preserving** or **TIP**.) A TIP procedure could be designed using the machinery of multi-valued SPFDs [17]. If one builds the multi-valued SPFD of circuit N_1^i and colors the corresponding SPFD graph, the resulting circuit N_2^i satisfies $N_1^i \leq N_2^i$ but has the same set of variables as N_1^i . So after a slight modification that takes into account the constraint function D_{inp} , multi-valued SPFDs could be used to design a TIP procedure. However we believe that another approach (which we are currently working on) similar to the one of [9] has a much better chance to succeed. (A more detailed discussion of pros and contras of different TIP procedures is beyond the scope of this report.) Importantly, once a TIP procedure is developed, LS_TI becomes a very powerful method of logic synthesis.

Let N_1^i be a subcircuit N_1 to be re-synthesized by LS_TI. Let N_1^i have k outputs and we want to build a subcircuit N_2^i of k outputs. The number of k -output Boolean functions f_2^i such that $f_1^i \leq f_2^i$, where f_1^i is the function implemented by N_1^i , is no less than $(2^k)!$. Indeed, even if some output assignments of N_1^i are unsatisfiable, there is always subcircuit N_2^i (if its number of inputs of N_2^i is greater or equal to k) such that all 2^k output assignments of N_2^i are satisfiable and $N_1^i \leq N_2^i$. Let this be the case and f_2^i be the function implemented by N_2^i . Any permutation of 2^k output assignments in the truth table of f_2^i gives a new function $f_2^{\prime i}$ such that $f_1^i \leq f_2^{\prime i}$.

The value of $(2^k)!$ is huge even for small k (e.g. if $k=5$, then $(2^5)!$ is equal to $2.6 \cdot 10^{35}$). So even if $\text{Spec}(N_1)$ has a very small granularity, LS_TI still enjoys great flexibility. On the other hand, since the complexity of LS_TI is linear in the number of subcircuits in $\text{Spec}(N_1)$, LS_TI is scalable (if one keeps the granularity of $\text{Spec}(N_1)$ small).

The idea of imposing restrictions on logic synthesis to make an optimized circuit verifiable has been studied previously (for example, in [1]). The problem is that, in general, such restrictions make a circuit verifiable at the expense of optimization quality. Interestingly, LS_TI not only produces a verifiable circuit but also has the potential to surpass current logic synthesis procedures in quality.

VI. COMPLEXITY OF EXTERNAL EQUIVALENCE CHECKING OF CIRCUITS WITH COMMON SPECIFICATION

Let N_1 be a circuit with specification $Spec(N_1)=\{N_1^1, \dots, N_1^k\}$. Let N_2 be a circuit with specification $Spec(N_2)=\{N_2^1, \dots, N_2^k\}$ produced from N_1 by either by EC_TE or EC_TI. In this section, we discuss the complexity of external equivalence checking of N_1 and N_2 .

A. Equivalence checking of circuits produced by LS_TE

In [9], a top commercial tool was used for “external” equivalence checking of circuits with a common specification (Table 2 of [9]). These results showed that even for circuits with a common specification of small granularity, their equivalence checking was too hard for that tool (even with a 10 hour time limit). On the other hand, all the examples were solved within 1-2 minutes by a program implementing EC_TE.

One can always pick circuits N_1 and N_2 with a common specification that will “break” current equivalence checking algorithms. The reason is that an external checker inevitably makes implicit assumptions that can be easily broken. For example, algorithms based on computing cut-points make an assumption that N_1 and N_2 have functionally equivalent internal points. However, if N_2 is produced from N_1 by LS_TE, N_1 and N_2 , in general, have no functionally equivalent points. Algorithms based on BDD computation make an implicit assumption that N_1 and N_2 have a small width while LS_TE can be used for optimization of circuits of arbitrary width. Equivalence checking based on recursive learning [12] assumes that implications relating points of N_1 and N_2 can be obtained inductively by a computation of small “recursion depth”. This assumption can be easily broken as well. The method of [13] also makes a breakable assumption that N_1 and N_2 do not have a large number of reconvergent fan-outs.

In terms of proof sizes (computed with respect to a concrete proof system like resolution), the problem with existing (and most likely any) external equivalence checkers is as follows. It may well be the case that any proofs of equivalence different from the ones generated by LC_TE are much “longer”. On the other hand, to find a proof generated by LC_TE one needs to build partitions $Spec(N_1)$ and $Spec(N_2)$, which is very hard. The reason is that finding a pair of subcircuits N_1^i, N_2^i that are toggle equivalent requires testing $\approx |N_1|^{p_1} * |N_2|^{p_2}$ pairs of subcircuits where $p_i, i=1,2$ is the granularity of $Spec(N_i)$.

B. Equivalence checking of circuits produced by LS_TI

Even though external verification of circuits built by LS_TE looks infeasible, verification of circuits produced by LS_TI is “even harder”. The reason is as follows. Suppose that N_2 with specification $Spec(N_2)$ is produced from N_1 with specification

$Spec(N_1)$ by LS_TE. Suppose an external equivalence checker somehow managed to find subcircuits N_1^i and N_2^i that are toggle equivalent. Then, it has to decide whether this toggle equivalence is “accidental” or N_1^i and N_2^i are subcircuits of $Spec(N_1)$ and $Spec(N_2)$. If N_1^i and N_2^i are toggle equivalent “accidentally”, then one cannot use outputs of N_1^i and N_2^i as “cut-points” to find subcircuits that are toggle equivalent in terms of previous cut-points (because the wrong choice of cut-points leads to false negatives). However, it is conceivable that toggle equivalence of subcircuits of N_1 and N_2 is a “rare” occasion and so N_1^i and N_2^i are subcircuits of $Spec(N_1)$, $Spec(N_2)$ with some reasonable probability.

The situation with LS_TI is vastly different. If N_2 with specification $Spec(N_2)$ is obtained from N_1 with specification $Spec(N_1)$ by LS_TI, any method of finding $Spec(N_1)$ and $Spec(N_2)$ faces huge false negative problem. Indeed, if $N_1^i \leq N_2^i$ holds for some subcircuits of N_1 and N_2 , then $N_1^i \leq N_2^i$ also holds if N_2^i is a subcircuit of N_2^i such that the outputs of N_2^i form a cut of N_2^i . (A cut of N_2^i cannot toggle “less” than the set of outputs of N_2^i .) Besides, $N_1^i \leq N_2^i$ also holds if N_2^i is a subcircuit of N_2^i and the set of outputs of N_2^i contains all the outputs of N_2^i (adding more outputs to N_2^i only increases toggling.) In other words, the number of pairs of subcircuits N_1^i, N_2^i for which $N_1^i \leq N_2^i$ holds is, in general, astronomical. So picking the “right” pair of subcircuits N_1^i, N_2^i is extremely unlikely and hence finding $Spec(N_1)$ and $Spec(N_2)$ looks even “more impossible” than in the case of LS_TE.

VII. CONCLUSION

In this report, we discuss how “external” equivalence checkers can be affected by the appearance of new powerful logic synthesis procedures. Our results imply that the increasing power of synthesis procedures may make external equivalence checking problematic if not impossible.

REFERENCES

- [1] P.Ashar, A. Gupta, S. Malik. *Using complete-1-distinguishability for FSM equivalence checking*. ICCAD 1996, pp. 346-353.
- [2] C.L.Berman. *Circuit width, register allocation, and ordered binary decision diagrams*. IEEE Trans. on CAD. Vol 10:8, 1991, pp. 1059-1066.
- [3] C.L.Berman, L.H.Trevillyan. *Functional comparison of logic designs for VLSI circuits*. ICCAD-89, pp.456-459.
- [4] D.Brand. *Verification of large synthesized designs*. ICCAD-93, pp.534-537.
- [5] R.Bryant. *Graph-Based Algorithms for Boolean Function Manipulation*. IEEE Transactions on Computers, Vol. C - 35, No. 8, August, 1986, pp. 677 - 691.
- [6] J.R.Burch, V.Singhal. *Tight integration of combinational verification methods*. ICCAD-98, pp.570-576.
- [7] R. Drechsler and S. Horeth. *Gatecomp: Equivalence Checking of Digital Circuits in an Industrial Environment*, International Workshop on Boolean Problems, pp. 195-200, 2002.
- [8] *Electronic Design Automation For Integrated Circuits Handbook*, by L.Lavagno, G.Martin, and L.Scheffer, Volume 2, Chapter 4, *Equivalence Checking*, by F. Somenzi and A. Kuehlmann.
- [9] E.Goldberg. *On Equivalence Checking and Logic Synthesis of Circuits with a Common Specification*. Proceedings of GLSVLSI, Chicago, April 17-19, 2005, pp.102-107 (<http://eigold.tripod.com/papers/glsvlsi-2005.pdf>).

- [10] A. Gupta, P.Ashar. *Integrating a Boolean Satisfiability Checker and BDDs for Combinational Equivalence Checking*. VLSI Design 1998. pp. 222-225.
- [11] A.Kuehlmann, F.Kroh. *Equivalence checking using cuts and heaps*, DAC-98, pp.263-268.
- [12] W.Kunz, D.Pradhan. *Recursive Learning: A New Implication Technique for Efficient Solutions to CAD-problems: Test, Verification and Optimization*. IEEE transactions on CAD, Vol. 13, No. 9, pp. 1143-1158, 1994.
- [13] H.Kwak, I.Moon, J.Kukula, T.Shiple. *Combinational equivalence checking through function transformation*, ICCAD-2002, pp. 526-533.
- [14] I.-H.Moon, C. Pixley. *Non-miter-based Combinational Equivalence Checking by Comparing BDDs with Different Variable Orders*. FMCAD 2004, 144-158
- [15] J. Moondanos, C.-J. H. Seger, Z.Hanna, D. Kaiss. *CLEVER: Divide and Conquer Combinational Logic Equivalence VERification with False Negative Elimination*. CAV-2001,pp. 131-143.
- [16] E.M. Sentovich et. al. *SIS: A system for sequential circuit synthesis*. Technical report, University of California at Berkeley, 1992. Memorandum No. UCB/ERL M92/41.
- [17] S.Sinha , S.Khatri, R.Brayton, A. Sangiovanni-Vincentelli. *Binary and multi-valued SPFD-based wire removal in PLA networks*, ICCD-2000, pp. 494-503.
- [18] E.Goldberg. K.Gulati. *Toggle equivalence preserving logic synthesis*. Technical Report. CDNL-TR-2005-0912, September, 2005. Available at <http://eigold.tripod.com/papers/tepls.pdf>