Toggle Equivalence Preserving (TEP) Logic Optimization

Eugene Goldberg
Cadence Design Systems
egold@cadence.com

Kanupriya Gulati
Texas A& M University
kanu.gulati@gmail.com

Sunil Khatri
Texas A&M University
sunilkhatri@tamu.edu

Abstract

We describe a procedure (called the TEP procedure) that, given a multi-output circuit M, builds another multi-output circuit M^* that is toggle equivalent to M. The TEP procedure can be used in the following two scenarios. First, since for single-output circuits toggle equivalence means functional equivalence, the TEP procedure can be used in "regular" logic synthesis. Second, the TEP procedure enables a powerful synthesis method called LS_TE (Logic Synthesis preserving Toggle Equivalence). Given a circuit N and its partitioning into subcircuits N_i, LS_TE builds an optimized circuit N^* by replacing subcircuits N_i with their toggle equivalent counterparts N^*_i. The replacement of N_i with N^*_i is done by the TEP procedure. We give results of optimizing single-output circuits by the TEP procedure and some preliminary results of using the TEP procedure in LS_TE. These results show the promise of the TEP procedure and LS_TE.

1. Introduction

In [4], a new method of logic synthesis was introduced. We refer to this method as LS_TE, which stands for Logic Synthesis preserving Toggle Equivalence. As shown in Figure 1, assume that a partitioning of N into subcircuits N_i, i=1, 2,…, k is specified. The main idea of LS_TE is to optimize N by replacing each subcircuit N_i with a toggle equivalent counterpart N^*_i, i=1,2,..,k.

Let us consider how LS_TE works by the example of Figure 1 where circuit N is partitioned into four subcircuits N_1,..,N_4. First, subcircuits N_1 and N_2 are replaced with their toggle equivalent counterparts N^*_1 and N^*_2. Then the relations $CF(N_1, N^*_1)$ and $CF(N_2, N^*_2)$ betweens outputs of N_1 and N^*_1 and N_2 and N^*_2 are computed. (These relations are called correlation functions (CF)). Then a single-output subcircuit N^*_3 that is toggle equivalent to the single-output subcircuit N_3 under the constraints specified by $CF(N_1, N^*_1)$ and $CF(N_2, N^*_2)$ is built. Since toggle equivalence for single output circuits means functional equivalence (modulo complement), outputs y_1 and y^*_1 are functionally equivalent (modulo complement). Finally single-output subcircuit N_4 is replaced with a single-output toggle equivalent subcircuit N^*_4, which makes outputs y_2 and y^*_2 functionally equivalent (modulo complement). So the optimized circuit N^* consisting of subcircuits N^*_1,..,N^*_4 is functionally equivalent to N modulo complement of its outputs.

The advantage of LS_TE is twofold (at least). First, LS_TE is *scalable*. The complexity of LS_TE is *linear* in the number of subcircuits N_i and exponential in the size of the largest subcircuit N_i or N^*_i, i=1,..,k. Since the number of subcircuits toggle equivalent to N_i is huge even if N_i is very small, LS_TE can explore a very large search space and still have *linear complexity*. Second, LS_TE *can escape local minima* that would trap a solution obtained by a traditional logic synthesis procedure (Section 3).

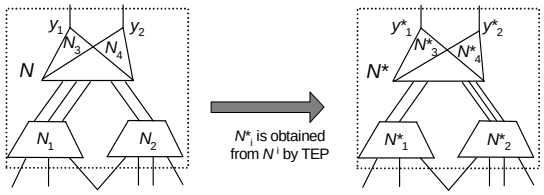

Figure 1. Optimization of circuit N by LS_TE

Unfortunately, [4] did not provide a specific procedure that, given a subcircuit N_i, would build a toggle equivalent subcircuit N^*_i. *The main contribution of this* paper is the introduction of such a procedure which we refer to as the Toggle Equivalence Preserving (TEP) logic optimization procedure. In the TEP procedure, we use a non-trivial convergence scheme that makes this procedure structure-agnostic. That is if a circuit M' toggle equivalent to an original circuit M is built by the TEP procedure, the topology of M' is not limited to that of M. This important feature of TEP is discussed in Section 3.

In the formulation of LS_TE given in [4], circuit N to be optimized is *partitioned* into subcircuits N_i, i=1,..,k However, LS_TE can be also applied if, for example, subcircuits N_i *share* internal gates. Suppose, for instance, that subcircuits N_3 and N_4 of Figure 1 share internal gates. Then when building subcircuit N^*_4 toggle equivalent to N_4, one can reuse the logic of N^*_3 (assuming that N^*_3 was synthesized before N^*_4). Such logic sharing can be done by the TEP procedure (slightly modified). However, a discussion of this topic is beyond the scope of this paper.

As we mentioned above, for single-output Boolean functions, toggle equivalence is the same as functional equivalence modulo negation. So, besides enabling LS_TE, the TEP procedure can be used in traditional logic synthesis. In order to compare the TEP procedure with traditional logic synthesis, the focus in this paper is on optimization of single-output functions. Even though the vast optimization flexibility of the TEP procedure can not be invoked for single-output functions, it still has the advantage of being structure-agnostic. As a consequence, for many single-output circuits the TEP procedure found better solutions than SIS [9]. Our initial results also show that for multiple output circuits (where the vast optimization flexibility can be exploited), the LS_TE procedure gave significant improvements over SIS

The rest of this paper is organized as follows. Section 2 discusses related previous work, including a comparison and contrasting of LS_TE and TEP with SPFDs [1][10][12]. In Section 3, we emphasize some important features of LS_TE and TEP procedure. Section 4 provides definitions. Section 5 details our TEP procedure. In Section 6, we report results of our experiments. Section **Error! Reference source not found.** concludes the paper, with some directions for future work in this topic.

2. Previous work

Multi-level logic synthesis can be performed using algebraic means such as factorization [2], kernelling [2][11] etc. Although these techniques are fast, being algebraic, they explore only a limited portion of the optimization space. Other techniques like ODC [6][7] and CODC [8] perform don't care based optimization, but they do not modify the structure of the circuit. (Sometimes a node gets removed as a result of don't care based optimization. However, such an occurrence is rare.) Toggle equivalence is different from the algebraic techniques, since it explores the "Boolean" options in the search space, while it differs from multi-level don't care based techniques since it does not restrict itself to the original circuit topology.

Sets of Pairs of Functions to be Distinguished (SPFDs) were introduced in [1][10][12] as a new way to do logic optimization. One should distinguish between SPFDs as a means to express circuit flexibility and concrete *methods* for computing SPFDs. The main difference between LS_TE and the method for computing SPFDs of [10][12] is that LS_TE is scalable. The method for computing SPFDs of [10][12] is unscalable because SPFDs are built by computing *non-local* relations between points of the circuit. Besides, when computing SPFDs by the method of [10][12] one has to follow the circuit topology. On the other hand, computations in LS_TE are *local* because they involve only subcircuits N_i and N^*_i (and correlation functions relating their inputs). Besides, LS_TE preserves only the high-level structure of the circuit (because subcircuits N_i and N^*_i are connected in the same way) but the topology of subcircuits N_i and N^*_i can be vastly different.

The "language" of SPFDs is sufficient to express the notions of toggle implication and equivalence that we use in the paper. For example, to test that circuits M and M' are toggle equivalent one can build SPFDs of M and M' and check them for graph isomorphism. However, toggle equivalence of M and M' can be computed much more efficiently without building their SPFDs (by performing two SAT-checks). Moreover, the formulation of the TEP procedure in terms of SPFDs is hard at best. An SPFD is a relation between input assignments while the TEP procedure operates on output assignments and the same pair of output assignments (i.e. the same toggle) may be caused by an *exponential* number of pairs of input assignments. In a sense, the problem is that the definition of SPFDs was tailored to facilitate their computation from outputs to inputs, while in LS_TE and TEP procedure computations go in the opposite direction.

3. Importance of LS_TE and the TEP procedure

In this section, we emphasize two important features of LS_TE and the TEP procedure. In Subsection 3.1, we show that LS_TE, in terms of equivalent transformations, can make moves that increase the size of intermediate circuits. This allows LS_TE to escape local minima that would trap a solution built by a traditional method of logic synthesis. In Subsection 3.2 we discuss the

importance of the novel convergence scheme of the TEP procedure.

3.1 Escaping local minima in LS_TE

Given a circuit N, a typical synthesis transformation is to replace a multi-output subcircuit N' of N with a *functionally equivalent* subcircuit N'' such that $|N''| < |N'|$. (Here $|M|$ is the size of circuit M.) The size of N' is kept small for complexity reasons. Suppose there is no transformation decreasing the size of N such that $|N'| < p$. This means that circuit N is stuck in a local minimum. To escape this minimum, one needs to make equivalent transformations that affect subcircuits of N larger than p. But how does one make such transformations in a scalable manner?

LS_TE answers the question above. Let N be partitioned into subcircuits $N_1,..,N_k$. By replacing subcircuits N_i, $i=1,.,k$ with toggle equivalent counterparts N^*_i LS_TE makes a *single* equivalent transformation that may encompass the entire circuit N (then the subcircuit N' we replace with an equivalent one is N itself). If the size of subcircuits N_i is small, this transformation can be done efficiently. Note that LS_TE can optimize N even if $|N_i| < p$, $i=1,.,k$. The reason is that replacement of N_i with N^*_i is *not an equivalent transformation*. So LS_TE can get N out of a local minimum even by making transformations of "small scope".

Suppose N implements the expression $x^2 < 100$ as shown in Figure 2. Here subcircuit N_1 implements $\boldsymbol{y}=square(\boldsymbol{x})$ and subcircuit N_2 implements $\boldsymbol{y} < 100$. (Let assume that the number n of bits in x is small enough to be handled efficiently.) LS_TE can optimize N as follows. First N_1 is replaced with an optimized subcircuit N^*_1 toggle equivalent to N_1 (e.g. N^*_1 may implement the function $abs(\boldsymbol{x})$ which is the simplest function toggle equivalent to $square(\boldsymbol{x})$). Then output relation $CF(N_1,N^*_1)$ is computed (as described in [4]). After that a subcircuit N^*_2 toggle equivalent to N_2 under constraint $CF(N_1, N^*_1)$ is built. (If N^*_1 implements $\boldsymbol{y}^*=abs(\boldsymbol{x})$, then N^*_2 implements $\boldsymbol{y}^* < 100$ modulo negation.) Note that N can not be optimized much by replacing N_1 with a *functionally equivalent* subcircuit N^*_1 (for example, N_1 can be an optimal implementation of $square(\boldsymbol{x})$). At the same time, LS_TE can dramatically optimize N because it can replace N_1 with *a toggle equivalent* subcircuit.

The replacement of N_1 with N^*_1 can be "simulated" as an equivalent transformation as shown in Figure 2 (on the right). Here R^*_1 is a re-encoding circuit such that $N_1 = R^*_1(N^*_1)$. (The second step of LS_TE is "simulated" as replacing N^*_1 and R^*_1 with N^*_2.) Note that even though N_1^* is much smaller than N_1 it may be the case that $|N_1| < |N^*_1|+ |R^*_1|$. In other words, the reason why LS_TE can escape local minima is that it may make transformations that *temporarily* increase the circuit size. (A discussion of this topic can be found in [5].)

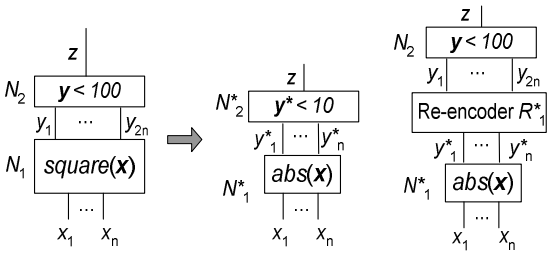

Figure 2. Optimization of expression $x^2 < 100$ by LS_TE

3.2 Novel convergence scheme of the TEP procedure

As we mentioned above, the importance of the TEP procedure is due to its enabling LS_TE. However, the TEP procedure is also important in its own right. Given a single-output circuit N, the TEP procedure can build a functionally equivalent circuit N^* with a *completely different topology*. (So it can be used in "regular" logic synthesis without any relation to LS_TE.) This property is extremely important for at least three reasons. First, N may not have any topology to reuse (e.g. if N is specified as the truth table or is represented implicitly). Second, N may contain some non-local redundancy, which makes reusing its topology unreasonable. Third, one may need to implement N using a particular library of gates (e.g. in technology mapping) and the current topology of N may be not good for these library.

In the current synthesis methods, if the topology of N can not be reused for some reason, a new circuit N^* is obtained from a *very limited space of implementations* (N^* may be further optimized using local transformations). For example, in SIS [9], if N is represented as the truth table, first, a circuit N^* equivalent to N is synthesized as a sum-of-products (which is a very limited class of circuits). Then by local transformations a multi-level circuit is obtained from N^*. (Another approach would be to build a circuit N^* of multiplexers (i.e. build a BDD [3]) equivalent to N and then optimize it using some local transformations. BDDs is another example of a restricted class of circuits.)

The reason why current methods have to restrict the class of implementations considered when changing the topology of N is the "convergence problem". Suppose we build a circuit N^* that does not use the topology of N. Then we have to make sure that the network of gates being built "converges" to a circuit equivalent to N. The TEP procedure solves this problem by introducing a very simple and general convergence scheme. Namely, it builds a sequence of circuits N^1, N^2,… such that a) N^{i+1} toggles strictly less than N^i and b) every circuit of this sequence toggles at least as much at the original circuit N. Here N^1 is an "empty circuit" consisting only of inputs of N. In other words, the TEP procedure builds a sequence of circuits that monotonically lose toggles until a circuit N^m toggle equivalent to N is built. The TEP procedure also restricts the class of implementations it considers since it requires that only primary outputs of N^i are allowed to feed the gates of N^{i+1} that are not in N^i. However, this is a mild restriction in comparison to ones used by existing methods. So, the TEP procedure can select an optimized implementation from a very general class of multi-level circuits.

4. Preliminaries and terminology

In this section, we recall the notion of toggle equivalence and its properties. All the propositions given in this paper are either proven in [4], or can be easily derived from them.

4.1 Toggle equivalence of Boolean functions

Definition 1. Let $\boldsymbol{f}:\{0,1\}^n \rightarrow \{0,1\}^m$ be an m-output Boolean function. Then, given $\boldsymbol{y'} = \boldsymbol{f}(\boldsymbol{x'})$ and $\boldsymbol{y''} = \boldsymbol{f}(\boldsymbol{x''})$, the pair $(\boldsymbol{y'}, \boldsymbol{y''})$ is ***a toggle*** if $\boldsymbol{y'} \neq \boldsymbol{y''}$.

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

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

Proposition 2. Let $\boldsymbol{f_1}$ and $\boldsymbol{f_2}$ be toggle equivalent single output Boolean functions. Then $\boldsymbol{f_1} = \boldsymbol{f_2}$ or $\boldsymbol{f_1} = \sim\boldsymbol{f_2}$**.**

Definition 3. Let N be a circuit. Let Y be the set of all variables of N. Let ***Sat*(*N*)** be the CNF expression for N, such that $Sat(N)=1$ iff the assignment $\boldsymbol{y}$ to Y is consistent within the circuit N. For example, if N consists of just one AND gate $w = x_1 \wedge x_2$, then $SAT(N) = (\sim x_1 \vee \sim x_2 \vee w) \wedge (x_1 \vee \sim w) \wedge (x_2 \vee \sim w)$.

Proposition 3. Let N_1 and N_2 be two toggle equivalent circuits, with variables Y_1 and Y_2 respectively. Let the output variables of N_1 and N_2 be Z_1 and Z_2 respectively. Then the function $H^*(Z_1,Z_2)$ specifying the one-to-one mapping H between the output vectors produced by N_1 and N_2 can be obtained from $Sat(N_1) \wedge Sat(N_2)$ by existentially quantifying away the variables of $(Y_1 \cup Y_2) \setminus (Z_1 \cup Z_2)$. (Then $H^*(\boldsymbol{z_1}, \boldsymbol{z_2}) = 1$ iff there is an input vector $\boldsymbol{x}$ such that $N_1(\boldsymbol{x}) = \boldsymbol{z_1}$ and $\boldsymbol{N}_2(\boldsymbol{x}) = \boldsymbol{z_2}$.)

4.2 Implication of toggling

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

Definition 4. Let $\boldsymbol{f_1}: \{0,1\}^n \rightarrow \{0,1\}^m$ and $\boldsymbol{f_1}: \{0,1\}^n \rightarrow \{0,1\}^k$ respectively be two m-output and k-output Boolean functions with the same set of input variables. Toggling of $\boldsymbol{f_1}$ ***implies toggling*** of $\boldsymbol{f_2}$ iff for any pair of input variable assignments $\boldsymbol{x'}$ and $\boldsymbol{x''}$, $\boldsymbol{f_1}(\boldsymbol{x'}) \neq \boldsymbol{f_1}(\boldsymbol{x''}) \Rightarrow \boldsymbol{f_2}(\boldsymbol{x'}) \neq \boldsymbol{f_2}(\boldsymbol{x''})$.

Definition 5. Let $\boldsymbol{f_1}$ and $\boldsymbol{f_2}$ be multi-output Boolean functions. Toggling of $\boldsymbol{f_1}$ ***strictly implies toggling*** of $\boldsymbol{f_2}$ if toggling of $\boldsymbol{f_1}$ implies toggling of $\boldsymbol{f_2}$ and there is a pair of assignments $\boldsymbol{x'}$ and $\boldsymbol{x''}$ such that $\boldsymbol{f_1}(\boldsymbol{x'}) = \boldsymbol{f_1}(\boldsymbol{x''})$ while $\boldsymbol{f_2}(\boldsymbol{x'}) \neq \boldsymbol{f_2}(\boldsymbol{x''})$. We will denote by $\boldsymbol{f_1} \leq \boldsymbol{f_2}$ (respectively $\boldsymbol{f_1} < \boldsymbol{f_2}$) the fact that toggling of function $\boldsymbol{f_1}$ implies toggling of (respectively strictly implies toggling of) $\boldsymbol{f_2}$. Let circuits N_1 and N_2 implement functions $\boldsymbol{f_1}$ and $\boldsymbol{f_2}$ respectively. We will denote by $N_1 \leq N_2$ (respectively $N_1 < N_2$) the fact that $\boldsymbol{f_1} \leq \boldsymbol{f_2}$ (respectively $\boldsymbol{f_1} < \boldsymbol{f_2}$).

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

4.3 Testing for Implication of Toggling.

Let N_1 and N_2 be two Boolean circuits to be checked for implication of toggling. Let X be the set of input variables of N_1 and N_2, while Y_1 and Y_2 are

respectively the sets of variables of N_1 and N_2. Let Z_1 and Z_2 be the sets of output variables of N_1 and N_2 respectively. Also, assume N^*_1 and N^*_2 are copies of N_1 and N_2, with output variables Y^*_1 and Y^*_2 respectively, and input variables X^*. Then $N_1 \le N_2$ holds iff the function $S(N_1, N_2)$ is unsatisfiable, where $S(N_1, N_2) = SAT(N_1) \wedge SAT(N_2) \wedge SAT(N^*_1) \wedge SAT(N^*_2) \wedge (Y_1 \neq Y^*_1) \wedge (Y_2 = Y^*_2)$.

Based on this, we can make the following three comments. 1) To test if $N_1 \le N_2$, we simply test the satisfiability of $S(N_1, N_2)$. If it is unsatisfiable (i.e. a constant zero), we conclude that $N_1 \le N_2$. 2) If $S(N_1, N_2)$ is satisfiable, it means that there exists a pair of input vectors $\boldsymbol{x}$ and $\boldsymbol{x}^*$ for which circuit N_1 toggles, while N_2 does not. 3) Let $S(N_1, N_2)$ be satisfiable. If we removed all toggles from N_1 that "are not in" N_2, we would have $N_1 \le N_2$. In other words, given two circuits N_1 and N_2, we can define a function *find_toggle_setdifference*(N_1, N_2) = *ALLSAT*$(S(N_1, N_2))$ which returns toggles of N_1 that are not matched by toggles of N_2. This is the set of toggles that must be removed from N_1. If the resulting set *ALLSAT*(N_1, N_2) is too large, its manageable *subset* can be used.

From Proposition 4, it follows that checking for toggle equivalence reduces to two satisfiability checks (henceforth called ***SAT checks***).

4.4 Correlation function

In this section, we briefly introduce the notion of correlation function, to extend definitions of toggle implication and toggle equivalence to the case when functions $\boldsymbol{f_1}$ and $\boldsymbol{f_2}$ have different sets of input variables.

Definition 6. Let X and Y be two disjoint sets of Boolean variables (the number of variables in X and Y may be different). A function $CF(X, Y)$ is called a ***correlation function*** if there are subsets $S^X \subseteq \{0,1\}^{|X|}$ and $S^Y \subseteq \{0,1\}^{|Y|}$ such that $CF(X, Y)$ specifies a bijective mapping $M: S^X \rightarrow S^Y$. Namely, $CF(\boldsymbol{x}, \boldsymbol{y})=1$ iff $\boldsymbol{x} \in S^X$, $\boldsymbol{y} \in S^Y$ and $\boldsymbol{y} = M(\boldsymbol{x})$.

Definition 7. Let Boolean functions $\boldsymbol{f_1}$ and $\boldsymbol{f_2}$ have different sets of variables (X and Y respectively) that are related by a correlation function $CF(X, Y)$. $\boldsymbol{f_1}$ and $\boldsymbol{f_2}$ are said to be ***toggle equivalent under input constraint*** $CF(X_1,Y)$, if for any pairs $(\boldsymbol{x}, \boldsymbol{y})$ and $(\boldsymbol{x}', \boldsymbol{y}')$ of input vectors such that $CF(\boldsymbol{x}, \boldsymbol{y})= CF(\boldsymbol{x}', \boldsymbol{y}')=1$, it is true that $\boldsymbol{f_1}(\boldsymbol{x}) \neq \boldsymbol{f_1}(\boldsymbol{x}') \Leftrightarrow \boldsymbol{f_2}(\boldsymbol{y}) \neq \boldsymbol{f_2}(\boldsymbol{y}')$. (Definition of toggle implication can be reformulated in a similar manner).

In LS_TE, the output relation between toggle equivalent subcircuits N_i and N^*_i is computed by existentially quantifying from $SAT(N_i) \wedge SAT(N^*_i) \wedge$ *Constr*(*inp_vars*(N_i),*inp_vars*(N^*_i)) all but output variables of N_i and N^*_i [4]. If N_i and N^*_i are subcircuits of the first topological level (and so have identical sets of input variables), then *Constr*(*inp_vars*(N_i), *inp_vars*(N^*_i)) just describes equivalence of corresponding variables. Since toggle equivalence of N_i and N^*_i means one-to-one mapping between output assignments, their output relation is a correlation function. In general, *Constr*(*inp_vars*(N_i), *inp_vars*(N^*_i)) is the conjunction of correlation functions that are output relations of all the subcircuits N_j, N^*_j feeding N_i, N^*_i. For the sake of simplicity, in Section 5, when describing the TEP procedure, we assume that circuit N_1 and its toggle equivalent counterpart N_2 have *identical sets of variables*.

5. TEP procedure

The TEP procedure produces the circuit N_2 (given a combinational circuit N_1) in a topological manner from inputs to outputs. These operations are illustrated in

Figure 3. The circuit N_2 is built up as a sequence of circuits $N_2^1, N_2^2, \ldots, N_2^m$. Each circuit N_2^i specifies a cut C_i of N_2 consisting of the primary outputs of N_2^i. In this way, the sequence of cuts C_i that are produced, are *topologically ordered***.** This means that for a pair of cuts C_i and C_p such that $i < p$ no path from a primary input to a primary output of N_2 can traverse C_p before C_i, although C_i and C_p may have common nodes. Then, if a node in C_p toggles for a given pair of input vectors, then there must be at least one node in C_i that toggles as well. So just from the fact that C_i and C_p are topologically ordered it follows that $N_2^p \le N_2^i$.

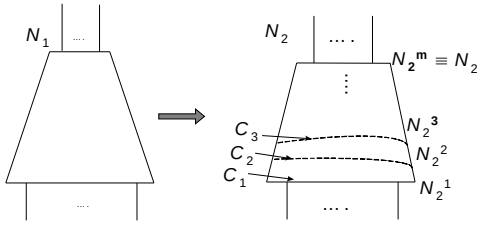

Figure 3. Sequence of circuits N_2^i constructed by TEP

The TEP procedure starts with $N_2^1 = \varnothing$ i.e. with an *empty* circuit which allows all possible toggles. As a result, $N_1 \le N_2^1$ (which is trivially true since the set

of inputs forms a cut of N_1). At each successive step, N_2^{i+1} is created from N_2^i such that $N_2^{i+1} < N_2^i$. The *invariant* that the TEP procedure maintains at each step is $N_1 \leq N_2^{i+1} < N_2^i$. In other words, the TEP procedure selectively removes one or more toggles in each step, until it is true that $N_2^m \leq N_1$. At this step, since $N_1 \leq N_2^m$, N_2^m is toggle equivalent to N_1, and the procedure returns the circuit N_2^m.

```
TEP(N1)
{if (is_constant(N1)) return "constant" ;
 N2^current = ∅;
 while(true)
  {if (N2^current ≤ N1) return N2^current ;
   N2^current = discard_toggles(N2^current, N1);
   N2^current = remove_redundant_outputs(N2^current);
  }}
```

Figure 4. Pseudocode of the TEP procedure

It is not hard to see that the TEP procedure has the desirable property of convergence. Since N_2^1 has all toggles, and $N_2^{i+1} < N_2^i$, the sequence of circuits N_2^1, N_2^2,…, N_2^m *must converge* to a circuit that is toggle equivalent to N_1.

The pseudocode of the TEP procedure is shown in Figure 4. To start with, we test if the input circuit N_1 is a constant, in which case the TEP procedure reports this fact. The sequence of circuits N_2^i mentioned earlier is built in the while loop. This sequence starts with an empty circuit $N_2^{current}$, which allows all possible toggles. In the while loop, we first check if $N_2^{current} \leq N_1$. If so, $N_2^{current}$ is toggle equivalent with N_1 (since $N_1 \leq N_2^{current}$ by construction) and we return $N_2^{current}$ as the resulting circuit N_2. If $N_2^{current} \leq N_1$ does not hold, then a new circuit $N_2^{current}$ is generated, such that it has at least one less toggle than the previous $N_2^{current}$. This operation is performed by the function *discard_toggles*, which is described in the next subsection. Finally, redundant outputs of $N_2^{current}$ are removed in the function *remove_redundant_outputs*. An output of $N_2^{current}$ is redundant if, after its removal from $N_2^{current}$, the condition $N_1 \leq N_2^{current}$ still holds.

Note that for each test for implication of toggling (i.e each "≤"check), we utilize the SAT-based algorithm described in subsection 4.3.

5.1 Discard toggles from N_2^i

Figure 5 describes the pseudocode of the *discard_toggle*s procedure used by the TEP procedure (Figure 4). The procedure *discard_toggles* consists of two parts. The procedures *remove_toggles* and *add_toggles* are explained in detail in the following subsections. In both these procedures, toggle removal and addition is done with AND gates, with their inputs appropriately complemented.

```
discard_toggles(N2^current, N1)
{ R* = find_toggle_setdifference(N2^current, N1);
  (N2^temp, R) = remove_toggles(R*, N2^current);
  D = find_toggle_setdifference(N1, N2^temp);
  N2^new_current = add_toggles(R, D, N2^current, N2^temp);
  return N2^new_current ;}
```

Figure 5. Pseudocode of the *discard_toggles* procedure

The *find_toggle_setdifference*($N_2^{current}$, N_1) routine was sketched in subsection 4.3. The heuristics of *remove_toggles* and *add_toggles* are aimed at minimizing the size of N_2.

5.1.1 Procedure *remove_toggles*. The function *remove_toggles* adds an AND gate G to $N_2^{current}$, to remove at least one toggle in the set R^* that is computed in line 1 of the *discard_toggles* procedure. (R^* specifies either the complete set of additional toggles that are present in $N_2^{current}$ and are not required in N_1 or a *manageable subset* of this set.) The resulting circuit is called N_2^{temp}, and the set of toggles of R^* actually removed are referred to as R.

Recall that each circuit N_2^i specifies a cut C_i of N_2 (consisting of the primary outputs of N_2^i.) Suppose the circuit $N_2^{current}$ specifies the cut $C^{current}$. Then the AND gate G above may have as its inputs, any of the nodes on $C^{current}$. After the addition of the AND gate G, the new cut C^{new} is formed from $C^{current}$ by a) adding to $C^{current}$ the node corresponding to the output of G; b) eliminating from $C^{current}$ the nodes that are *toggling* inputs of G.

Suppose the cut $C^{current}$ consists of the set of nodes Y. Suppose that $r = (\boldsymbol{y}, \boldsymbol{y'})$ is a toggle from the set R^*. Let Y_1 and Y_2 form a partition of Y, such that Y_1 (Y_2) corresponds to the components of $\boldsymbol{y}$ and $\boldsymbol{y'}$ which are different (same). In other words, Y_1 (Y_2) corresponds to the nodes of Y that have different (same) values for the toggle $r = (\boldsymbol{y}, \boldsymbol{y'})$.

To remove the toggle r, we add an AND gate G. We consider two cases.

Case i): If $Y_1 = 1$, then gate G has two inputs. One of these inputs is specified by the variable of Y_1, and another input is chosen from Y_2. All possible polarities of the second input are considered as well. The configuration for which $G(\boldsymbol{y}) = G(\boldsymbol{y'})=0$ and that removes *the largest number* of toggles of R^* is selected.

Case ii): if $Y_1 > 1$, then gate G has $|Y_1|$ inputs. These inputs are connected to the variables in Y_1, with appropriate polarity selection to guarantee that $G(\boldsymbol{y}) = G(\boldsymbol{y}') = 0$.

In both cases, the construction of gate G guarantees that $G(\boldsymbol{y})=G(\boldsymbol{y}')=0$. After adding the gate G, we form the cut C^{new} by removing from $C^{current}$ all the nodes in Y_1 and adding the output of G. Then, the toggle $r = (\boldsymbol{y}, \boldsymbol{y}')$ is removed from the nodes of C^{new}. The circuit resulting from this operation is called N_2^{temp}.

5.1.2 Procedure *add_toggles*. Unfortunately, adding the gate G (described in the previous subsection) may sometimes remove certain toggles that are required in N_1. As a consequence, we have to perform a "clean-up" step, and add these toggles back into the design.

We begin with computing D, the set of toggles that need to be added. D is computed by *find_toggle_setdifference*(N_1, N_2^{temp}). The objective is to add minimum number of AND gates that re-introduce all toggles from D, and at the same time minimize the number of toggles that get re-introduced from R. It is not hard to prove that one can always re-introduce a toggle from the set D, by using a 2-input AND gate H, with appropriately selected inputs and input polarities, without re-introducing a toggle from the set R. The proof is omitted due to space constraints.

Once again, we have two cases to consider, analogous to those in the previous subsection:

Case i): When the gate G added by *remove_toggles* was a 2-input gate, with $|Y_1| = 1$, then one of the inputs of H is the same as the node in Y_1. The other input of H is selected from among nodes in Y_2. All possible nodes and polarities are explored *to maximize* the weighted cost function n_1+p*n_2. Here, n_1 is the number of toggles of R prevented from being re-introduced, n_2 is the number of toggles of D re-introduced and p is the weight parameter (that was set to 1 in our experiments). We add only those gates for which n_1 is 1 or more.

Case ii): If $|Y_1| > 1$, then select the first input of H from Y_1, and the second from Y, except the input already chosen as the first leg. The cost function to select inputs and their polarities is identical to the one explained in Case i above.

After each AND gate added to the circuit, the set D is recomputed. The routine *add_toggles* continues to add AND gates until the set D reduces to $\varnothing$. At this point the resulting circuit $N_2^{new\_current}$ is returned. It satisfies the property that $N_1 \leq N_2^{new\_current} < N_2^{current}$. Note that for a single gate added in *remove_toggles*, zero, one or more AND gates could be added in the following call of *add_toggles*.

6. Experimental results

Our preliminary implementation of the TEP procedure is in SIS [9]. We performed various experiments to compare TEP with traditional logic synthesis commands. The experiments were performed on a 3 GHz Xeon CPU, with 2GB of memory.

Table 1. Results for optimizing arithmetic expressions

Exper	#bits	script.rugged		collapse, script.rugged		TEP		BDD
		time (s)	#gates	time (s)	#gates	time (s)	#gates	time (s)
$x^2 < C$	10	3	590	0.5	28	5	20	0.01
$x^2 < C$	14	34	1,361	95	44	17	21	0.25
$x^2 < C$	16	94	1,808	2,151	52	54	46	0.9
$x^2 < C$	27	35	7,037	>10h	-	282	50	Mem
$x^2 < C$	30	56	8,681	>10h	-	525	57	Mem
$C_1*x < C_2$	16	24	1,054	121	15	14	19	0.07
$C_1*x < C_2$	18	39	1,201	1,659	17	25	28	0.11
$C_1*x < C_2$	38	37	6,709	>10h	-	497	58	Mem
$C_1*x < C_2$	50	136	10,483	>10h	-	2,183	66	Mem

Table 1 provides the results of applying TEP procedure and SIS for optimizing circuits implementing the expressions $x^2 < C$ and $C_1*x < C_2$ for different word sizes. (In contrast to the example of Subsection 3.1, the expressions above were optimized as *one circuit* i.e. by one call of the TEP procedure.) In all experiments, the value of C was chosen to be 200 (the results do not change much if one varies C). C_1 and C_2 were set to decimal value 11111. The two expressions above can be reduced to much simpler expressions $x < C'$ and $x < C''$ respectively where C' is equal to *sqrt*(C) and C'' is equal to C_2/C_1. The objective of this experiment was to show that since TEP is structure-agnostic it can be used to simplify *"non-local" redundancy*. Note that although optimization of these expressions can be easily done manually, one can give examples of non-local redundancies that are much harder to find manually or by a program. Any logic synthesis procedure that changes the original circuit's structure locally (like SPFDs or don't care based optimizations) can easily get trapped in a local minimum. Note that only for smaller values of C, C_1 and C_2, it is possible to build ROBDDs. For the experiments in Table 1, we set the threshold of R^* at 10 as explained in sections 5.1 and 4.3 i.e. R^* contained only 10 (out of a huge number of) toggles

to be removed. The reason why the TEP procedure worked so well with such a small subset R^* was that by adding an AND gate to remove a toggle of R^* *explicitly*, we may *implicitly* remove a huge number of toggles that were "skipped" in R^*.

The first column in Table 1 represents the expression being simplified, while the second column represents the word size. Columns 3 and 4 represent the runtime and number of gates returned by *script.rugged*. Columns 5 and 6 represent the runtime and number of gates returned by *collapse* followed by *script.rugged*. The corresponding results for TEP are provided in Columns 7 and 8, while Column 9 represents the time taken to build a ROBDD (using the *nanotrav* package in CUDD). The notation "Mem" indicates a memory out condition. In all cases, the number of gates refers to the number of gates required after optimization and decomposition using AND2 and inverter gates.

We observe that the *script.rugged* requires significantly more gates than TEP. This is because *script.rugged* performs only local changes of the circuit and so SIS gets stuck in a local minimum. TEP, on the other hand, uses only the functionality of the circuit and so produces a dramatically smaller circuit. We may run *collapse* before *script.rugged*, to allow SIS to re-structure the logic better. However for all but the smallest word widths, *collapse* fails. Similarly, the ROBDD computation fails for large word widths, while TEP optimizes these circuits with less than 66 gates. Interestingly, the arithmetic expressions we used turned out to have "local redundancies" (however, in general, global redundancy of a circuit does not "translate" into local redundancies). So redundancy removal in SIS [9] can optimize them with comparable results by taking about two orders of magnitude more time than the TEP procedure.

Table 2 shows the results of running a commercial tool (CT) on circuits produced by *script.rugged* and the TEP procedure. We used single-output circuits extracted from MCNC benchmarks. The objective of the experiment was to show that even for very small circuits, TEP can achieve better optimization. (The other reason for targeting small subcircuits is that in LS_TE, the TEP procedure is used for optimizing subcircuits N^i of circuit N that are assumed to be *small*.) The first column of Table 2 shows names of circuits and the output number (in parentheses). The second column provides the number of inputs in the single output circuits. Columns 3 and 4 provide the mapped area and delay for the output of *script.rugged* mapped by CT, while Columns 5 and 6 provide these numbers for the TEP output mapped by CT. The standard cell library had 38 gates, implemented in a 0.18μ process. The licensing agreement for CT requires us not to identify its name. The results of Table 2 indicate that TEP based circuits, after mapping, result in a 12.5% area improvement, and a 1.6% delay penalty over circuits optimized with *script.rugged* before mapping with CT. The TEP results improve on the *script.rugged* results for 85% of the examples in terms of area, and for 45% of the examples in terms of delay.

The objective of the experiment summarized in Table 3 was to provide a brief demonstration of the ability of LS_TE. The LS_TE method was used to optimize two-stage circuits. Both stages correspond to standard benchmark circuits, with the second stage being a single output circuit. The outputs of the first stage are inputs to the second. MCNC benchmarks *rd84* and *squar5* were used as the first stage circuits. The second stage circuits are single-output circuits extracted from MCNC benchmarks (second column). Columns 3 through 6 give the number of gates in optimized circuits and runtimes for optimization by *script.rugged* and LS_TE.

Table 2. Optimization of single-output circuits

Circuits	#in-puts	*script.rugged*→CT		TEP→CT	
		area	delay(ps)	area	delay(ps)
b12(3)	4	83.635	77	62.727	73
i5(37)	5	130.679	75	114.999	106
s_opt(6)	3	151.589	102	151.588	87
pm1(10)	8	182.952	120	156.815	109
squar5(1)	5	250.905	118	156.815	105
misex2(15)	5	177.725	128	156.816	113
x4(34)	8	224.771	124	172.497	142
x3(64)	5	250.906	110	224.769	121
5xp1(5)	4	308.405	163	229.996	164
squar5(3)	5	491.356	169	235.223	143
i7(10)	5	282.268	126	235.224	146
apex7(35)	8	360.675	149	245.678	165
b9(1)	7	282.270	134	245.679	150
ttt2(7)	5	224.769	131	250.905	121
apex1(43)	8	266.587	110	256.134	129
apex6(51)	7	392.040	178	277.042	183
ttt2(4)	6	266.586	136	297.950	127
i7(28)	6	444.312	161	308.405	142
qpcle(4)	8	392.040	160	423.402	139
sqrt8ml(3)	8	3183.359	652	2299.966	584

When optimizing a circuit N of Table 3, TEP is used twice. We first replace the stage 1 circuit N_1 with its toggle equivalent counterpart N^*_1, using TEP. After this the correlation function relating outputs of N_1 and N^*_1 is computed as described in [4]. (One

needs to compute the correlation function because N_1 is a multi-output circuit.) Using the correlation function, the second stage circuit N_2 is replaced with a toggle equivalent counterpart N^*_2, using TEP a second time. The composition of circuits N^*_1 and N^*_2 form a circuit N^* functionally equivalent to N modulo negation. Since we assume that N_1 and N_2 were designed independently, any output encoding for N_1 is in a sense as good as the original one. So the heuristics of TEP that aim at finding a toggle equivalent counterpart of N_1 that is as small as possible makes sense.

Note that the number of gates resulting from TEP optimization is significantly smaller than for SIS. In fact, on average, TEP requires 50.5% fewer gates than *script.rugged*. Our current TEP implementation is unoptimized, and we have efforts underway to improve the runtimes of TEP.

Table 3. Optimization of two-stage circuits by LS_TE

stage 1	stage 2	*script.rugged*		TEP	
		# gates	time(s)	# gates	time(s)
rd84	5xp1(5	138	0.8	53	62
rd84	alu2(5)	78	0.5	47	62
rd84	b12(3)	101	0.6	37	62
squar5	alu4(1)	43	0.1	23	3.4
squar5	b12(2)	42	0.1	20	2.7
squar5	c8(11)	28	0.1	17	2.2

7. Conclusions

We have presented a new toggle equivalence preservation based procedure (TEP) for logic synthesis. This TEP procedure can be used in the scenario shown in Figure 1. The idea is to re-synthesize a circuit N (consisting of subcircuits N_i), in a manner that the high-level partitioning structure of N is retained. Each subcircuit N_i is re-synthesized into a design N^*_i , using the TEP procedure. This re-synthesis explores a huge optimization flexibility since the outputs of N_i are re-encoded by TEP. This TEP procedure was formulated for multi-output circuits. The TEP procedure is structure-agnostic, unlike existing logic optimization procedures. Also, it is able to explore all possible output encodings efficiently during synthesis. For single-output circuits, toggle equivalence is the same as functional equivalence modulo negation. Therefore, we tested TEP on single-output circuits, to enable a fair comparison with existing synthesis approaches, although the full power of TEP is exhibited for multi-output circuits. The preliminary implementation of TEP is done in SIS, using a SAT-based computation. First results show encouraging improvements over SIS When the full power of TEP is utilized (for multi-output circuits) we expect yet further improvements.

8. References

[1] R.Brayton, *Understanding SPFDs: A new method for specifying flexibility*. In Proc. of the International Workshop on Logic Synthesis (Tahoe City, CA), May 1997.

[2] R.Brayton and C.McMullen. *The Decomposition and Factorization of Boolean Expressions*. In Proc. IEEE International Symposium on Circuits and Systems, pp.49-54, May. 1982.

[3] R.Bryant. *Graph-Based Algorithms for Boolean Function Manipulation*. IEEE Trans. on Computers, Vol. C - 35, No. 8, August, 1986, pp. 677 - 691.

[4] E.Goldberg. *On equivalence checking and logic synthesis of circuits with a common specification.* GLSVLSI, Chicago, April 17-19, 2005, pp.102-107, http://eigold.tripod.com/papers/glsvlsi-2005.pdf.

[5] E.Goldberg *Escaping Local Minima in Logic Synthesis*, IWLS-2007, San Diego 2007.

[6] H. Savoj and R.Brayton. *The Use of Observability and External Don't Cares for the Simplification of Multi-Level Networks*. DAC,1990, pp.297-301.

[7] H. Savoj, R.Brayton, and H.Touati. *Extracting Local Don't Cares for Network Optimization*, ICCAD,1991, pp.514-517.

[8] H.Savoj. *Don't Cares in Multi-Level Network Optimization*. PhD thesis, University of California Berkeley, Electronics research laboratory, May 1992.

[9] 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.

[10] S.Sinha, R.K.Brayton. *Implementation and use of SPFDs in optimizing Boolean networks*. ICCAD-1998, pp. 103-110.

[11] J.Vasudevamurthy and J.Rajski. *A Method for Concurrent Decomposition and Factorization of Boolean Expressions*, ICCAD,1990, pp.510-513.

[12] S.Yamashita, H.Sawada, A.Nagoya. *A new method to express functional permissibilities for LUT based FPGAs and its applications.* ICCAD,1996, pp.254-261.