Escaping Local Minima in Logic Synthesis

Eugene Goldberg (Cadence Berkeley Labs)

Abstract. In this paper, we continue studying Logic Synthesis Preserving Specification (LSPS). Given a combinational circuit N and its partition into subcircuits N_1,.., N_k (this partition is called a *specification* of N), LSPS optimizes N by replacing each subcircuit N_i with toggle equivalent subcircuit N^*_i. As we showed before, LSPS is *scalable*. In this paper, we demonstrate that LSPS can be also viewed as an elegant way to address the *local minimum entrapment problem*. The latter remains a thorny issue for the heuristic algorithms for solving hard combinatorial problems. In this paper, we give only theoretical arguments in favor of LSPS. The preliminary experimental results of LSPS can be found in [6][7].

1. Introduction

When solving hard computational problems one has to address the problem of local minima entrapment. Due to the huge size of the search space, a typical algorithm A for solving, say, an NP-hard problem uses heuristics specifying *small* changes to be made to the current solution. In such an algorithm, a change is accepted if it improves a cost function. (Henceforth, we assume that one needs to *minimize* a cost function.) This leads the current solution to a local minimum. The latter is the situation when in the set of moves used by A no move can improve the cost of the current solution. Unfortunately, the quintessential feature of NP-hard problems is that a local minimum can be arbitrarily deep. This means that to get the current solution out of a local minimum by the moves allowed in A, one may have to make an *unbounded number* of moves that make the cost the solution higher.

Unfortunately, making moves increasing the cost function dramatically increases the search space. So an algorithm making such moves has no chance to converge to a better solution in a reasonable time. For example, in the optimization method called simulated annealing (its application to logic synthesis is given in [2]), the number of moves increasing the cost function is controlled by a "cooling" schedule. The smaller the temperature is, the less likely it is that such a move is accepted in simulated annealing. If the cooling schedule becomes sufficiently long, simulated annealing can reach the global minimum (and so get out of any local minimum). Unfortunately, these schedules may take the time even larger than that of just enumerating all possible solutions.

A typical logic synthesis procedure (being a special case of an optimization problem) also suffers from the local entrapment problem mentioned above. Usually, when optimizing a circuit N, such a procedure generates a sequence of circuits $N^1, N^2,\ldots$, (where $N^1=N$) such that N^{i+1} is functionally equivalent to N^i and $cost(N^{i+1}) < cost(N^i)$. (For the sake of simplicity, henceforth, we assume that $cost(N^i)$ is the number $|N^i|$ of gates in N^i.) For complexity reasons, the transformations used by such a procedure are local and affect only a small part of the circuit. Eventually, a circuit N^m of this sequence gets stuck in a local minimum.

In this paper, we show that logic synthesis preserving specification (LSPS) introduced in [3][4] actually suggests an interesting approach to the local minimum entrapment problem. (The site *http://eigold.tripod.com/papers.html* contains all referenced papers co-authored by the author of this paper.) Let N be a single output circuit to be optimized and N_1,.., N_k be a partition of N into subcircuits. (In this paper we, assume, unless otherwise stated, that one needs to optimize a *single-output* circuit N. A discussion of applying LSPS to multi-output circuits can be found in [8].) This partition is called a *specification* of N. The idea of the method of [3][4] is to modify N by replacing subcircuits N_i with *toggle equivalent* subcircuits N^*_i that are optimized according to the required cost function. Then the circuit N^* consisting of subcircuits N^*_i is functionally equivalent to N (modulo negation) and has the same specification as N (because subcircuits N^*_i are connected with each other in N^* exactly as subcircuits N_i in N). In this paper, we show that a single transformation performed by LSPS can be represented as k *functionally equivalent* transformations of the original circuit each of which *may increase* the size of the current circuit. So LSPS can be viewed as a logic synthesis procedure that performs equivalent transformations going "against" the cost function. This means that, in general, transformations of LSPS *can not be reproduced* by a "traditional" logic synthesis procedure *monotonically* reducing circuit size at every step and performing "local" transformations.

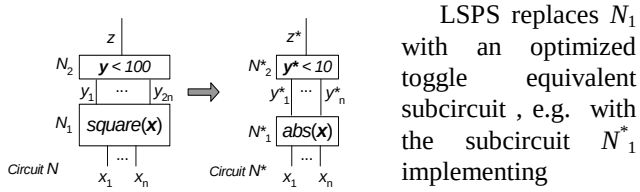
This paper is structured as follows. An example of LSPS is given in Section 2. In Section 3, we recall the basic notions of toggle equivalence and correlation function and describe LSPS of [4]. The recent developments in LSPS are given in Section 4. Section 5 relates LSPS to existing synthesis procedures from the viewpoint of enabling equivalence checking procedures. Section 6 analyzes LSPS from the optimization point of view. In Section 7, we discuss two kinds of optimization performed by LSPS. Section 8 gives reasons for LSPS to be successful. Some conclusions are made in Section 9.

2. Example

Suppose one needs to optimize a single-output circuit N implementing the arithmetic expression $x^2 < 100$ as shown

in Figure 1. Circuit N consists of subcircuits N_1 and N_2 connected as a cascade. (In general, LSPS can handle the case when the connections of subcircuits N_i in N are described by an arbitrary directed acyclic graph.) The subcircuit N_1 implements the function $\boldsymbol{y}$=s*quare*($\boldsymbol{x}$) and N_2 implements the function $\boldsymbol{y} < 100$.

It is not hard to see that the expression $x^2 < 100$ can be replaced with much simpler expression $abs(x) < 10$. Below we show how this optimization can be done by LSPS. (This simplification may look "trivial" and so doable by a high-level optimizer. However, one can easily modify this example in such a way that high-level optimization becomes much less trivial if not impossible.)



LSPS replaces N_1 with an optimized toggle equivalent subcircuit , e.g. with the subcircuit N^*_1 implementing

Figure 1. Optimization of $x^2 < 100$ by LSPS

$\boldsymbol{y}^* = abs(\boldsymbol{x})$. Then it computes relation $D_{out}(N_1, N^*_1)$ specifying the bijective mapping between the output assignments produced by N_1 and N^*_1. (As we showed in [4], if two circuits are toggle equivalent, there is a one-to-one mapping between output assignments these circuits produce. Note that N_1 has twice the number of outputs of N^*_1.) After that, a subcircuit $N^*_2(\boldsymbol{y}^*)$ is constructed that is toggle equivalent to $N_2(\boldsymbol{y})$ (implementing $\boldsymbol{y} < 100$) under the input constraint specified by $D_{out}(\boldsymbol{y}, \boldsymbol{y}^*)$). In the case N^*_1 implements $\boldsymbol{y}^* = abs(\boldsymbol{x})$, subcircuit N^*_2 has to implement $\boldsymbol{y}^* < 10$ (or its negation). For single-output circuits, toggle equivalence means functional equivalence (modulo negation) [4]. So N and the circuit N^* composed of N^*_1 and N^*_2 are functionally equivalent (modulo negation).

3. Logic synthesis preserving common specification

In this section, we recall definitions of toggle equivalence and correlation function and describe the procedure LSPS of [4].

3.1 Toggle equivalence

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

Definition 2. Let $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ be two Boolean functions of the same set of variables. Functions $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ are called ***toggle equivalent*** if $\boldsymbol{f}_1(\boldsymbol{x}) \neq \boldsymbol{f}_1(\boldsymbol{x}') \Leftrightarrow \boldsymbol{f}_2(\boldsymbol{x}) \neq \boldsymbol{f}_2(\boldsymbol{x}')$. (Note that $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ may have different number of outputs.) Circuits N_1 and N_2 implementing toggle equivalent functions $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ are called ***toggle equivalent circuits***.

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 $f^* = f(\ldots, x_i=0,\ldots) \vee f(\ldots, x_i=1,\ldots.)$.

Definition 4. Let N be a circuit. Denote by $v(N)$ the set of variables of N. Denote by $\boldsymbol{Sat(v(N))}$ the Boolean function such that $Sat(\boldsymbol{h})=1$ iff the assignment $\boldsymbol{h}$ 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 $Sat(v(N)) = (\sim x_1 \vee \sim x_2 \vee y) \wedge (x_1 \vee \sim y) \wedge (x_2 \vee \sim y)$.

Proposition 1. [4] Let N_1 and N_2 be toggle equivalent and Z_1, Z_2 be the sets of their output variables. Let function $K^*(Z_1, Z_2)$ be obtained from $Sat(v(N_1)) \wedge Sat(v(N_2))$ by existentially quantifying away the variables of N_1 and N_2 except those of $Z_1 \cup Z_2$. The function $K^*(Z_1, Z_2)$ implicitly specifies the one-to-one mapping K between output vectors produced by N_1 and N_2. Namely, $K^*(\boldsymbol{z_1}, \boldsymbol{z_2}) =1$ iff $\boldsymbol{z_1}=K(\boldsymbol{z_2})$.

3.2 Correlation function

In this section, we use the notion of a correlation function to extend definition of toggle equivalence to the case where functions $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ have different sets of variables.

Definition 5. 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 $Q^X \subseteq \{0,1\}^{|X|}$ and $Q^Y \subseteq \{0,1\}^{|Y|}$ such that $Cf(X,Y)$ specifies a bijective mapping M: $Q^X \rightarrow Q^Y$. Namely $Cf(\boldsymbol{x}, \boldsymbol{y})=1$ iff $\boldsymbol{x} \in Q^X$ and $\boldsymbol{y} \in Q^Y$ and $\boldsymbol{y} = M(\boldsymbol{x})$.

Informally, $Cf(X,Y)$ is a correlation function if it specifies a bijective mapping between a subset Q^X of $\{0,1\}^{|X|}$ and a subset Q^Y of $\{0,1\}^{|Y|}$.

Let $\boldsymbol{f}_1(X)$ and $\boldsymbol{f}_2(Y)$ be two multi-output Boolean functions where $X=\{x_1,\ldots, x_k\}$ and $Y=\{y_1,\ldots, y_p\}$ are their variables. Let $Cf(X, Y)$ be a correlation function relating variables of $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$. Then one can introduce notions of toggle equivalence as follows. Boolean functions $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ are said to be toggle equivalent, if for any two pairs $(\boldsymbol{x}, \boldsymbol{y})$ and $(\boldsymbol{x}', \boldsymbol{y}')$ of vectors such that $Cf(\boldsymbol{x}, \boldsymbol{y})=Cf(\boldsymbol{x}', \boldsymbol{y}')=1$, it is true that $\boldsymbol{f}_1(\boldsymbol{x}) \neq \boldsymbol{f}_1(\boldsymbol{x}') \Leftrightarrow \boldsymbol{f}_2(\boldsymbol{y}) \neq \boldsymbol{f}_2(\boldsymbol{y}')$.

The mapping between output vectors produced by toggle equivalent circuits N_1 and N_2 (implementing functions $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ respectively), can be obtained from $Sat(v(N_1)) \wedge Sat(v(N_2)) \wedge Cf(X,Y)$ by existentially quantifying away all the variables of $v(N_1) \cup v(N_2)$ except the output variables of N_1 and N_2.

3.3 Logic synthesis preserving specification

Let N be a single-output circuit. Denote by $Spec(N)$ a **specification** of N i.e. a partition of N into subcircuits $N_1,\ldots, N_k$. Following [4] we assume that specification $Spec(N)$ is topological. Let G be a directed graph whose nodes are subcircuits N_i and an edge of G directed from node N_i to node N_j implies that an output of N_i is

connected to an input of N_j. *Spec*(N) is called ***topological*** if G is acyclic. Since *Spec*(N_1) is topological, one can assign levels to subcircuits N_i. The pseudocode of LSPS of [4] is given in Figure 2. There, we assume that the numbering of subcircuits is *topological*. That is if $i < j$ then *topological_level*(N_i) ≤ *topological_level*(N_j). In other words, subcircuits N_i, i=1,..,k are processed by the LSPS procedure in topological order, from inputs to outputs.

Let us revisit the example of Section 2. LSPS starts with subcircuit N_1 (implementing *square*(***x***)) and recovers the function $D_{inp}(N_1, N^*_1)$ relating the inputs of N_1 and subcircuit N^*_1 to be built (line 3 of pseudocode). The inputs of N_1 are inputs of N (and so N_1 has the lowest topological level 1). In that case $D_{inp}(N_1, N^*_1) \equiv 1$. Then a subcircuit N^*_1 toggle equivalent to N_1 (e.g. implementing *abs*(***x***)) is synthesized (line 4). In the end of this iteration, the function $D_{out}(N_1, N^*_1)$ relating outputs of N_1 and N^*_1 is built (line 5) as described in Proposition 1. (That is $D_{out}(N_1, N^*_1)$ is obtained by existentially quantifying away from the expression $Sat(v(N_1)) \wedge Sat(v(N^*_1)$ all the variables but the output variables N_1 and N^*_1.) Since N_1 and N^*_1 are toggle equivalent, there is a one-to-one mapping between the output vectors they produce. So $D_{out}(N_1, N^*_1)$ is a correlation function.

```
1 LSPS(N, Spec(N),cost_function) {
2   for (i=1; i <= k ; i++) {
3     D_inp(N_i, N*_i)= constraint_function(N, N*,i);
4     N*_i = synth_toggle_equivalent(N_i, D_inp,cost_function)
5     D_out(N_i, N*_i) = exist_quantify(N_i,N*_i, D_inp); }
6 return(N*,Spec(N*))}
```

Figure 2. Pseudocode of LSPS procedure

In the next iteration, subcircuit N_2 is processed similarly to N_1 with one exception. The inputs of N_2 are fed by the outputs of N_1. Then the function $D_{inp}(N_2, N^*_2)$ relating inputs of N_2 and circuit N^*_2 (synthesized in line 4) equals $D_{out}(N_1, N^*_1)$. (In general, the inputs of a subcircuit N_i of *Spec*(N) are fed by outputs of more than one subcircuit N_j of *Spec*(N). To obtain $D_{inp}(N_i, N^*_i)$ one has to take the conjunction of $D_{out}(N_j, N^*_j)$ for all subcircuits whose outputs feed inputs of N_i and N^*_i. It is not hard to show that in this case $D_{inp}(N_i, N^*_i)$ is a correlation function too.)

Let N^*_2 be a subcircuit built by LSPS that is toggle equivalent to N_2. If N^*_2 does not have redundant outputs (i.e. outputs equal to each other modulo negation or implementing constants), it has only out output. Then N and the resulting circuit N^* (composed of subcircuits N^*_1 and N^*_2) are functionally equivalent modulo negation.

4. Recent developments in LSPS

In this section, we describe recent improvements to LSPS made in [5] , [6], [7] and [8].

4.1 Better complexity parameterization

In [3] and [4], the complexity of LSPS was given in the granularity of specification of circuit N. The **granularity** of *Spec*(N)={N_1,.., N_k} is the size $|N_i|$ of the largest subcircuit N_i of *Spec*(N) (in the number of gates). The complexity of LSPS is exponential in the granularity of N and linear in the number of subcircuits N_i of *Spec*(N). So, if, for example, the size of subcircuits of *Spec*(N) is bounded by a constant, the complexity of LSPS is linear.

The result above was improved in [5]. There, we considered an equivalence checking procedure for circuits N and N^* with a common specification (this procedure "enables" LSPS). We showed that the complexity of this procedure (and hence the complexity of LSPS) is exponential in the width of specifications *Spec*(N) and *Spec*(N^*) and linear in the number of subcircuits. The **width** of *Spec*(N) is $max(W_1,W_2)$. Here W_1 (respectively W_2) is the maximum number of outputs (respectively maximum circuit width) among the subcircuits N_i of *Spec*(N). (The first definition of circuit width was given in [1].)

Informally, the result of [4] means that the complexity of LSPS remains linear even if the size of subcircuits of *Spec*(N) and *Spec*(N^*) is not bounded but the number of outputs and width of subcircuits of *Spec*(N) and *Spec*(N^*) is bounded. So the width of *Spec*(N) provides a better parameterization of LSPS than granularity.

4.2 Logic synthesis preserving toggle implication

In [6], we introduced a generalization of LSPS based on the notion of toggle implication. In this subsection we will refer to the method of [4] as LS_TE and to the method of [6] as LS_TI. Here LS stands for logic synthesis, TI for toggle implication and TE for toggle equivalence.

Definition 6. Let f_1 and f_2 be two Boolean multi-output functions with the same set of variables X={x_1,…, 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 $f_2(x') \neq f_2(x'')$.

Let N be a single output circuit and *Spec*(N)={N_1,…, N_k}. We assume here that the numbering of subcircuits N_i is topological (as in Subsection 3.3). The idea of [6] is to replace the first k-1 subcircuits N_i with subcircuits N^*_i such that $N^*_i \leq N_i$. (Here "≤" denotes the fact that toggling of N^*_i is implied by toggling of N_i.) The last subcircuit of *Spec*(N) (i.e. subcircuit N_k) is replaced with N^*_k that is *toggle equivalent* to N_k. Then the circuit N^* composed of subcircuits N^*_1,…, N^*_k is functionally equivalent to N (modulo negation). In contrast to LS_TE, in LS_TI, when replacing subcircuit N_i, i=1,..,k-1 with subcircuit N^*_i (such that $N_i \leq N^*_i$) one has to impose the limit on the number of outputs in N^*_i. Otherwise, LS_TE just replaces N_i with an "empty" circuit N^*_i consisting only of inputs (because in this case $N_i \leq N^*_i$ holds).

In [4], we showed that Boolean functions f_1 and f_2 are toggle equivalent iff $f_1 \leq f_2$ and $f_2 \leq f_1$. So toggle implication is a more general relation than toggle equivalence. which makes LS_TI more powerful than LS_TE. Methods LS_TI and LS_TE can be viewed as two versions of LSPS. For the sake of clarity, in the following exposition we will use the version LS_TE of LS_PS. However, one can easily extend this exposition to LS_TI.

4.3 TEP procedure

The key part of LSPS is the procedure that, given a subcircuit N_i of *Spec*(N), builds an optimized circuit N^*_i that is toggle equivalent to N_i (under input constraints specified by $D_{inp}(N_i, N^*_i)$). Such a procedure (called Toggle Equivalence Preserving procedure or TEP procedure for short) was introduced in [7]. Introduction of the TEP procedure has made LSPS "a reality". Given a circuit N_i, the TEP procedure builds a sequence of circuits N_i^1, N_i^2,... where $N_i^1=N_i$ that converges to a circuit $N_i^m = N^*_i$ toggle equivalent to N_i. For each circuit N_i^p of this sequence, $N_i \leq N_i^p$ holds. So the TEP procedure can be also used for LS_TI (i.e. for logic synthesis preserving toggle implication). One just needs to stop the TEP procedure when the number of outputs in N_i^p is below a predefined threshold and use N_i^p as the subcircuit N^*_i replacing N_i.

4.4 Finding good specification

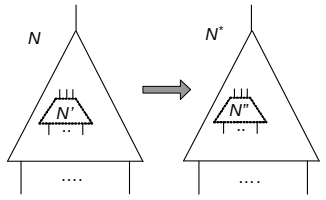
In [8], we consider the problem of finding a "good" specification of a circuit N. We conjecture that, in general, such specification can not be built efficiently because high-order relations have to be discovered. However, in the very important case of narrow circuits, a good specification has a trivial topology: a cascade of subcircuits. So a good specification of a narrow circuit can be found automatically. We also conjecture that a good specification of a wide circuit N (such as a multiplier) can be extracted from a natural partitioning of N into subcircuits (like adders give a natural partitioning of a multiplier).

In [8], we also extend the notion of specification to the case where subcircuits N_i of *Spec*(N) may *share gates*.

5. Relation of LSPS to existing synthesis methods

In this section, we relate LSPS to other methods of logic synthesis from the viewpoint of the enabling equivalence checking procedure i.e. at a very high level of abstraction. (A comparison of LSPS with SPFDs [9][10] can be found in [7].) In [8], we also compare LSPS with existing methods from two other angles: complexity and relation subcircuit/environment. In particular, we show that in contrast to existing methods, LSPS is based on the notion of *relative* (rather than *absolute*) circuit complexity. Besides, LSPS operates under the "*friendly environment paradigm*" (as opposed to the "*unfriendly environment*" approach employed by existing synthesis methods).

Any logic synthesis transformation has to have an enabling equivalence checking procedure that is used to certify the correctness of this transformation. In a typical logic synthesis transformation shown in Figure 3, a multi-output subcircuit N' of N is replaced with an optimized and *functionally equivalent* subcircuit N''. The corresponding enabling equivalence checking procedure consists of block-level and compositional parts. The block-level part (that is non-trivial) is to prove that N' and N'' are functionally equivalent. The compositional part is *trivial*. It just says that if one replaces subcircuit N' with a functionally equivalent subcircuit N'', the resulting circuit N^* is functionally equivalent to N.



LSPS is enabled by the equivalence checking procedure of [4] that has the *non-trivial compositional part*. In terms of enabling equivalence

Figure 3. A typical synthesis transformation

checking procedures, LSPS is a generalization of existing synthesis procedures. Indeed, replacing N' with a functionally equivalent subcircuit N'' is a special case of LSPS. (In this case *Spec*(N) consists of subcircuit N' and one-gate subcircuits corresponding to the gates of N that are not in N'. Since N' is replaced with a *functionally equivalent* subcircuit N'' there is no "re-encoding debt" in the form of the correlation function $D_{out}(N', N'')$. So one does not have to propagate this debt to the output of N and so does not have to change the logic fed by N'.)

Suppose, however, that a transformation of a traditional logic synthesis procedure changes the functionality of N' but the modified subcircuit N'' is toggle equivalent to N'. Suppose, for example, that this transformation is to replace a complex gate G' of N' with a simpler gate G'' such that this replacement is "unobservable" at the outputs of N'. Since the subcircuit N'' is not functionally equivalent to N', the replacement of G' with G'' is "observable". So this transformation will be rejected by a logic synthesis procedure enabled by the usual equivalence checking procedure (with the trivial compositional part). However, it is within the power of LSPS to accept the replacement of G' with G'' (because they are toggle equivalent) and re-synthesize the logic fed by N' to make the replacement of N' with N'' a correct transformation.

6. LSPS from optimization point of view

In this section, we consider LSPS from the optimization point of view. Namely, we show that LSPS can be

simulated by an algorithm performing small equivalent transformations that *may increase* the circuit size. On the one hand, this implies that, in general, LSPS performs transformations that *can not be reproduced* by a traditional logic synthesis procedure that a) monotonically reduces the circuit size and b) makes "local" transformations. On the other hand, this means that LSPS can escape local minima that trap solutions of traditional logic synthesis algorithms.

Intuitively, the *depth of local minima* LSPS can escape depends on the width of *Spec*(*N*). The deeper a local minimum is, the more coarse partitioning of *N* into subcircuits is necessary to avoid it. In particular, if *Spec*(*N*) consists of *N* itself, LSPS can potentially escape any local minimum (but the complexity of such escape is exponential in |*N*| and so prohibitively high).

The exposition in this section is structured as follows. In Subsection 6.1, we recall the problem of local minima entrapment in the context of traditional logic synthesis. Subsection 6.2 describes a modification of LSPS called LSPS$^+$. Since LSPS is a special case of LSPS$^+$, everything we say about LSPS$^+$ applies to LSPS as well. Subsection 6.3 shows that LSPS$^+$ can escape local minima that trap solutions of traditional synthesis methods.

6.1 Local minima entrapment

Let *N* be a circuit to be optimized. A typical synthesis procedure performs a sequence of transformations shown in Figure 3. Each transformation reduces the value of a cost function (as we mentioned above, in this paper we assume that *cost*(*N*)=|*N*|). Then a typical synthesis procedure builds a sequence of circuits $N^1, N^2,\ldots$, such that N^{i+1} is functionally equivalent to N^i and $|N^{i+1}| < |N^i|$. Eventually a circuit N^m gets stuck in a local minimum (that can be arbitrary far from a global minimum) and the synthesis procedure terminates. To escape a local minimum, a synthesis algorithm has to make a number of moves *increasing circuit size*. However, currently there are no efficient algorithms for doing this.

6.2 Modification of LSPS

In this subsection, we consider a modification of LSPS further referred to as LSPS$^+$. The pseudocode of LSPS$^+$ is shown in Figure 4. On the one hand, we use LSPS$^+$ to study LSPS from the optimization point of view. On the other hand, LSPS$^+$ can be actually used in practice as a more "flexible" version of LSPS. As we show below, LSPS can be viewed as a special case of LSPS$^+$. So everything we say about LSPS$^+$ *applies to LSPS as well.*

The main difference between LSPS$^+$ and LSPS is that LSPS$^+$ tries to compute a re-encoding circuit R^*_i such that $R^*_i(N^*_i)$ is functionally equivalent to N_i. (Here N^*_i is a subcircuit toggle equivalent to subcircuit N_i of *Spec*(*N*)) That is in addition to computing the relation $D_{out}(N_i,N^*_i)$, LSPS$^+$ also computes a circuit R^*_i "implementing" this relation. In contrast to LSPS, LSPS$^+$ can estimate the size of the current circuit even before replacing all subcircuits N_i of *Spec*(*N*). Hence, LSPS$^+$ can stop as soon as the size of the current circuit becomes smaller than the size of the original circuit *N*.

```
1  LSPS+(N, Spec(N),cost_function) {
2    for (i=1; i <= k ; i++) {
3      D_inp(N_i, N*_i)= constraint_function(N, N*,i);
4      N*_i = synth_toggle_equivalent(N_i, D_inp,cost_function)
5      D_out(N_i, N*_i) = exist_quantify(N_i,N*_i, D_inp);
6      if (simple(D_out(N_i, N*_i)) R*_i=re-encoder(D_out(N_i, N*_i));
7      else |R*_i| = ∞
8      if (|N*_1| +..+|N*_i| +|R*_p1| + .. + |R*_pi| < |N_1| + .. |N_i|)
9          return(N*,Spec(N*),R*_p1,…,R*_pi);}
10   return(N*,Spec(N*))}
```

Figure 4. Pseudocode of LSPS$^+$

Let us explain how LSPS$^+$ works by the example shown in Figure 5 where the circuit *N* to be optimized consists of subcircuits N_1 and N_2. At the first step of LSPS$^+$, the subcircuit N_1 is replaced with a toggle equivalent counterpart N^*_1 and the relation $D_{out}(N_1, N^*_1)$ is computed as in LSPS. However, in contrast to LSPS, if the relation $D_{out}(N_1, N^*_1)$ is "simple" enough, LSPS$^+$ computes a re-encoder R^*_1 (line 6 of Figure 4) such that $R^*_1(N^*_1(\boldsymbol{y}))$ is *functionally equivalent* to $N_1(\boldsymbol{y})$. (Let us assume, for the sake of clarity, that LSPS$^+$ considers relation $D_{out}(N_i, N^*_i)$ as "simple", if the number of outputs in N_i and N^*_i does not exceed a threshold value.) If $D_{out}(N_1, N^*_1)$ is "complex", then R^*_1 is not generated and the size of R^*_1 is set to infinity (line 7). Suppose that R^*_1 is actually built by LSPS$^+$ and $|N^*_1|+|R^*_1|<|N_1|$ (line 8). Then LSPS$^+$ stops here and generates the resulting circuit as a cascade of N^*_1, R^*_1, N_2.

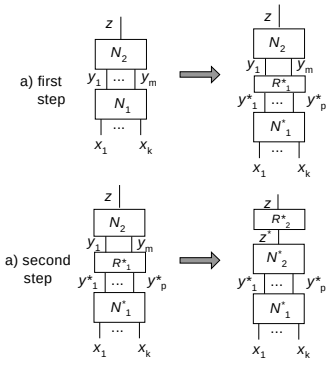


Figure 5. Example of LSPS$^+$ run

If $|N^*_1|+|R^*_1| \geq |N_1|$, then LSPS$^+$ computes N^*_2 that is toggle equivalent to N_1 under input constraint specified by $D_{out}(N_1, N^*_1)$. LSPS$^+$ also computes the re-encoder R^*_2 that just inverts the output of N^* if the latter is the negation of *N*. (Note that at this point R^*_1 "disappears" from the circuit. For that reason, in line 8 of Figure 4 we take into account only *some* of re-encoders generated by the *i*-th step. LSPS$^+$ "drops" re-encoder R^*_i as soon as each subcircuit N_s of *Spec*(*N*) fed by outputs of N_i is replaced

with a toggle equivalent subcircuit N^*_s. The re-encoders $R^*_{p1},.,R^*_{pi}$ of line 8 are the ones that have to be preserved by the *i*-th step.)

LSPS can be viewed as a special case of LSPS. Indeed, suppose that LSPS$^+$ considers the relation $D_{out}(N_i, N^*_i)$ as "complex" if the number of outputs in N_i, N^*_i is greater than one. Then none of the "internal" re-encoders R^*_i will be generated and $|R^*_i|$ will be set to infinity (assuming that all "internal" subcircuits N_i have more than one output). Only when LSPS$^+$ reaches a pair of corresponding primary outputs of N and N^*, it computes a trivial re-encoder (a buffer or an inverter). So, in this case, LSPS$^+$ behaves exactly as LSPS.

6.3 Escaping local minima by LSPS$^+$

Suppose that during the run of LSPS$^+$ shown in Figure 5, the final circuit N^* consists of N^*_1, N^*_2 and R^*_2 (if an inverter is necessary) and $|N^*| < |N|$. This means that although after the first step, LSPS$^+$ did not stop because $|N^*_1|+|R^*_1| \geq |N_1|$, eventually it managed to build a circuit N^* smaller than N. Inequality $|N^*_1|+|R^*_1| \geq |N_1|$ may hold for the following three reasons. First, the relation $D_{out}(N_1, N^*_1)$ is too complex and R^*_1 is not built by LSPS$^+$ (so $|R^*_1|$ is set to infinity). Second, even though there is a re-encoder $R^{*\prime}_1$ such that $|N^*_1|+|R^{*\prime}_1| < |N_1|$, the re-encoder R^*_1 built by LSPS$^+$ is larger than $R^{*\prime}_1$ and so $|N^*_1|+|R^*_1| \geq |N_1|$. Third, there is no re-encoder R^*_1 such that $|N^*_1|+|R^*_1| < |N_1|$. For example, this is the case when N_1 is an optimal circuit. (Note, that even if N_1 is optimal, the circuit N consisting of N_1 and N_2 may be arbitrary far from a global minimum).

The third case above is particularly interesting. It means that LSPS$^+$ may make transformations that increase the size of *intermediate* circuits. This implies that LSPS$^+$ (and hence LSPS) may make transformations that *can not be reproduced* by traditional synthesis algorithms. To be precise, transformations made by LSPS and LSPS$^+$, in general, are not reproducible by a synthesis algorithm that a) *monotonically* reduces the circuit size at every step and b) makes transformations that affect a subcircuit whose size is limited by the *granularity* of *Spec*(*N*). In other words, in general, a traditional procedure (trying to reduce circuit size at every step) may reproduce a transformation made by LSPS$^+$ only by increasing the *scope* of transformation. In the worst case, a transformation performed by LSPS can be reproduced only if the entire circuit N changes in *one* equivalent transformation.

7. Horizontal and vertical optimization

In Subsections 7.1 and 7.2 below we consider two complementary kinds of optimization performed by LSPS$^+$: horizontal and vertical. We use the term horizontal optimization to refer to the situation when optimization of N is due to re-synthesis of subcircuits N_i, N_m of *Spec*(N) that are *topologically independent.* (That is gates of N_i are not in the transitive fan-out of gates of N_m and vice versa.) Vertical optimization takes place when two *topologically dependent* circuits N_i and N_m are re-synthesized by LSPS$^+$ (For example, outputs of N_i may feed inputs of N_m.)

7.1 Horizontal optimization

Let *Spec*(N) of N have topologically independent subcircuits N_i, N_m with similar toggling behavior. Then N_i and N_m can be replaced with subcircuits N^*_i and N^*_m that share a lot of logic. (In the extreme case, when N_i and N_m are toggle equivalent, one can pick, say, N_i as both N^*_i and N^*_m, in other words, replace N_m with N_i.) We will refer to the case of optimization achieved due to sharing of logic by topologically independent subcircuits N^*_i and N^*_m as **horizontal optimization**.

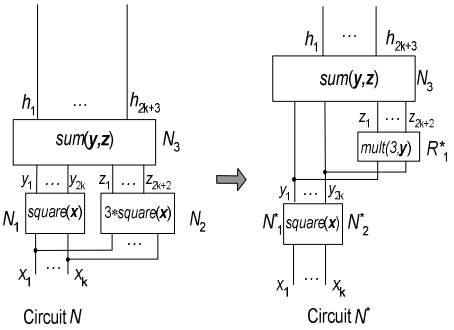


Figure 6. Example of horizontal optimization

An example of horizontal optimization is shown in Figure 6. The circuit N on the left implements the expression x^2+3*x^2. Here subcircuits N_1, N_2, N_3 of N implement functions $\boldsymbol{y}=square(\boldsymbol{x})$, $\boldsymbol{z}=3*square(\boldsymbol{x})$ and $sum(\boldsymbol{y},\boldsymbol{z})$ respectively. The circuit N^* on the right is obtained by LSPS$^+$. Subcircuit N_1 is replaced with subcircuit N^*_1 that is identical to N_1. Subcircuit N_2 is replaced with subcircuit N^*_2 also identical to N_1 (it is not hard to see that N_1 and N_2 are toggle equivalent so one can replace N_2 with N_1). Then LSPS$^+$ generates re-encoder R^*_1 implementing the function $\boldsymbol{z}=mult(3, \boldsymbol{y})$. Since R^*_1 is a fairly simple function, $|N^*_1| + |N^*_2| + |R^*_1| < |N_1| + |N_2|$ where $|N_1|=|N_2|=|N^*_1|$ and $|N^*_2| = 0$ and so LSPS$^+$ stops at this point.

7.2 Vertical optimization

Let us return to the example of Section 2. Application of LSPS$^+$ to this example is shown in Figure 7. LSPS$^+$ performs two steps. In the first step, the subcircuit N_1 implementing $square(\boldsymbol{x})$ is replaced with circuit N^*_1 implementing $abs(\boldsymbol{x})$ and re-encoder R^*_1. In the second step, re-encoder R^*_1 and circuit N_2 (implementing $\boldsymbol{y} < 100$) are replaced with subcircuit N^*_2 and re-encoder R^*_2 (implementing an inverter or a buffer). Subcircuit N^*_2 is picked to be toggle equivalent to $N_2(R^*_1(\boldsymbol{y}^*))$.

Obviously, the subcircuit N^*_1 implementing *abs*(**x**) is smaller than N_1 implementing *square*(**x**). Given a particular implementation N_1 of *square*(**x**), it is not clear if there is a re-encoder R^*_1 such that $R^*_1(N^*_1(\mathbf{x}))$ is equivalent to $N_1(\mathbf{x})$ and $|N^*_1| + |R^*_1| < |N_1|$. If, for example, N_1 is an optimal implementation of *square*(**x**), then obviously, there does not exist a re-encoder R^*_1 such that $|N^*_1| + |R^*_1| < |N_1|$. (Note that even if N_1 is an optimal implementation of *square*(**x**), the circuit *N* is very far from an optimum.)

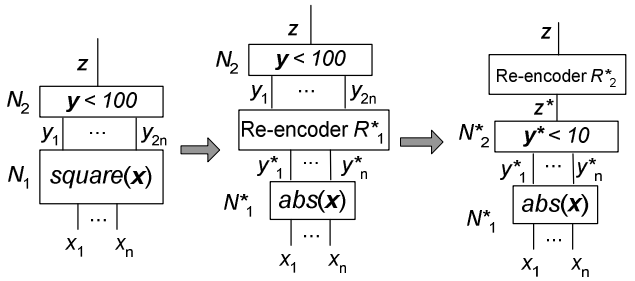


Figure 7. Vertical optimization by LSPS

A trivial re-encoder is the circuit N_1 itself (because *square*(*abs*((**x**)) = *square*(**x**)). However, in this case, obviously $|N^*_1| + |R^*_1| > |N_1|$. So LSPS$^+$ is able to build a circuit N^* that is much smaller than N_1 even though the intermediate circuit (which is the cascade of N^*_1, R^*_1 and N_2 is larger than the initial circuit *N*). We will refer to the case of optimization achieved due to "redistribution" of logic between topologically dependent subcircuits as **vertical optimization**.

8. Why should it work?

In this section, we discuss the reasons for LSPS$^+$ to succeed in circuit optimization. In Subsection 8.1, we show that LSPS$^+$ provides a framework for designing efficient algorithms escaping local minima. In the following subsections we give various aspects of LSPS$^+$ that should make it successful. In Subsection 8.2, we show that horizontal optimization is a natural way to share logic between "cooperating" logic blocks. Subsections 8.3 and 8.4 explain how LSPS$^+$ can get away with transformations increasing circuit size in vertical optimization. Namely, we show that vertical optimization can be successful due to loss of information in the original circuit. In case a circuit *N* has many more inputs than outputs, this loss of information is "global" (Subsection 8.3). However, even if *N* does not lose information globally or loses very "little", it still can have subcircuits that lose information locally (Subsection 8.4).

8.1 High-level view

LSPS$^+$ can be viewed as just a framework for studying and designing algorithms that that can escape local minima. Suppose we try to optimize a circuit *N* using a set of small equivalent transformations as shown in Figure 3. Suppose there is no transformation reducing the size of *N*, if $|N'| < p$ (i.e. if the size of the subcircuit N' of *N* we replace with N'' consists of less than *p* gates). This essentially means that *N* is stuck in a local minimum. To get *N* out of this minimum, one needs to make equivalent transformations that affect a subcircuit of *N* larger than *p*. But how does one make such transformations in a scalable manner?

LSPS$^+$ answers the question above. By replacing subcircuits N_i of *Spec*(*N*) with toggle equivalent counterparts N^*_i LSPS$^+$ makes a *single* equivalent transformation that may encompass the entire circuit *N* (in this case the subcircuit N' we replace with an equivalent one is *N* itself). If *Spec*(*N*) is narrow, this transformation can be done efficiently. If there are no "small" equivalent transformations optimizing *N*, some replacements of N_i of *Spec*(*N*) with N^*_i may increase the size of the intermediate circuit (i.e. $|N^*_i| + |R^*_i| > |N_i|$). Obviously, LSPS$^+$ can not guarantee that after replacing subcircuits N_i with toggle equivalent subcircuits N^*_i it will always obtain a smaller circuit N^*. Nevertheless, since a circuit trapped in a local minimum can be *arbitrary far* from the optimum, developing algorithms of escaping local minima is extremely important. LSPS$^+$ suggests an elegant way to cope with the problem of local minima entrapment.

8.2 Horizontal optimization

Before, we gave a *made-up* example of applying horizontal optimization successfully (Figure 6). However, there is a good reason to believe that horizontal optimization can be successfully used in practice. Suppose, for example, that a high-level specification contains two combinational blocks *A* and *B* that "cooperate" with each other. This cooperation means that when the output of *A* changes its value (in terms of multi-valued variables) *B* "almost always" changes its value too. In other words, *A* and *B* are almost toggle equivalent (in terms of multi-valued functions). Then one can pick encodings of output variables of *A* and *B* so that many outputs of *Impl*(*A*) and *Impl*(*B*) are functionally equivalent and so can be shared. (Here *Impl*(*C*) is an implementation of block *C*.)

In practice, however, when translating high-level descriptions, Boolean encodings are chosen arbitrarily. In such a case even though *Impl*(*A*) and *Impl*(*B*) are "almost" toggle equivalent, they may not share any (or share very little) logic. Then LSPS$^+$ can improve the situation by replacing *Impl*(*A*) and *Impl*(*B*) with toggle equivalent subcircuits that share a lot of logic. This can be done by a slightly modified TEP procedure of [7]. (A discussion of such modification is beyond the scope of this paper.)

8.3 Vertical optimization (global loss of information)

Let N be a circuit to be optimized. Let N have many more inputs than outputs. In this case, it inevitably loses information. Let $C_1,..,C_p$ be a topologically ordered set of cuts of N where C_1 is the set of inputs of N and C_p is the set of outputs of N. Let $\boldsymbol{x}$, $\boldsymbol{y}$ be a pair of input vectors such that $\boldsymbol{x} \neq \boldsymbol{y}$ and $N(\boldsymbol{x})=N(\boldsymbol{y})$. Then there should be a cut C_i, $i=2,\ldots,p$ such that $C_i(\boldsymbol{x})=C_i(\boldsymbol{y})$ and for every cut C_j, $j > i$ it is also true that $C_j(\boldsymbol{x})=C_j(\boldsymbol{y})$. In other words, loss of information means that as one moves from inputs to outputs, cuts C_i become less and less toggling.

By replacing a subcircuit N_i of $Spec(N)$ with N^*_i, LSPS$^+$ makes a temporary "re-encoding debt" in the form of $D_{out}(N_i, N^*_i)$. Since LSPS$^+$ replaces subcircuits of $Spec(N)$ in topological order, it "pushes" the debts in the direction of cuts that toggle less and less. Then it is possible that even though $|N^*_i| + |R^*_i| > |N_i|$ (but $|N^*_i| < |N_i|$), LSPS$^+$ still can succeed in optimizing N. The debt $D_{out}(N_i, N^*_i)$ that is too big to pay now, may eventually become much smaller.

Let us consider, for instance, the example of Section 2. By replacing N_1 implementing $square(\boldsymbol{x})$ with N^*_1 implementing $abs(\boldsymbol{x})$, LSPS$^+$ runs up a large "debt". However, since the circuit N (namely its subcircuit N_2 implementing $\boldsymbol{y} < 100$) loses a lot of information, LSPS does not have to pay this debt "in full". By replacing N_2 with a small subcircuit N^*_2 (implementing $\boldsymbol{y}' < 10$) LSPS$^+$ pays only a small fraction of this debt and nevertheless obtains circuit N^* functionally equivalent to N.

8.4 Vertical optimization (local loss of information)

Let N be a circuit to optimized. Suppose N does not lose (much) information globally (which implies that the number of inputs and outputs of N are comparable). The fact that N does not lose information globally does not mean that N can not lose information locally.

Let N' be a subcircuit N. Let $inp(N')$ and $out(N')$ denote the set of input and output variables of N' respectively. A variable v is in $inp(N')$ if it describes an input of a gate of N' fed by a gate that is not in N'. A variable v is in $out(N')$ if it describes the output of a gate of N' that feeds a gate that is not in N'. Suppose the size of $out(N')$ is much larger than that of $inp(N')$. Then one can apply LSPS$^+$ for optimization of N' (by partitioning N' into subcircuits and replacing these subcircuits with toggle equivalent counterparts). As we explained in Subsection 8.3, LSPS$^+$ may succeed because N' loses information (from the viewpoint of N this is a local loss of information).

Suppose, for example, that we need to optimize an implementation of a function $y=f(x)$ specified as follows. If $x^2 < 100$ then $y = f_1(x)$, otherwise $y = f_2(x)$. Let the expression $x^2 < 100$ be implemented as shown in Figure 1 (on the left). Then even if a circuit N implementing $f(x)$ preserves (almost) all information, the single-output subcircuit N' implementing $x^2 < 100$ loses a lot of information and can be optimized by LSPS$^+$ as described above.

9. Conclusions

In this paper, we consider various aspects of Logic Synthesis Preserving Specification (LSPS). We show that LSPS provides an elegant solution to the local minimum entrapment problem. Since the size of a circuit trapped in a local minimum can be arbitrarily far from the global minimum, the importance of addressing this problem is hard to overestimate.

References

[1] C.L.Berman. *Circuit width, register allocation, and ordered binary decision diagrams.* IEEE Trans. on CAD. Vol 10:8, 1991, pp. 1059-1066.

[2] P. Farm, E.Dubrova and A.Kuehlmann. *Logic Synthesis Using Simulated Annealing.* IWLS-2006, pp. 9-15.

[3] E.Goldberg. *Logic synthesis preserving high-level specification.* International Workshop on Logic Synthesis, IWLS-2004.

[4] 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

[5] E.Goldberg. *Equivalence checking of circuits with parameterized specifications.* International Conference on Theory and Applications of Satisfiability Testing, St Andrews, UK, June 19-23,2005, LNCS 3569, pp.107-121.

[6] E.Goldberg, K. Gulati. *On Complexity of External and Internal Equivalence Checking.* Technical Report CDNL-TR-2006-0105, January 2006.

[7] E.Goldberg, K.Gulati. *Toggle Equivalence Preserving Logic Synthesis.* Technical Report CDNL-TR-2005-0912, September 2005.

[8] E.Goldberg. *Escaping Local Minima in Logic Synthesis (and some other problems of logic synthesis preserving specification).* Technical Report CDNL-TR-2007-0212, February 2007.

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

[10] 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.