

Escaping Local Minima in Logic Synthesis (and some other problems of logic synthesis preserving specification)

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



CDNL-TR-2007-0212
February 2007

Eugene Goldberg (Cadence Berkeley Labs), egold@cadence.com

Abstract. In this report, we continue studying Logic Synthesis Preserving Specification (LSPS). Given a combinational circuit N and its partition into subcircuits N_1, \dots, 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 report, 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. We also discuss finding a “good” specification of a circuit. In particular, we show that for narrow circuits there is a natural specification subcircuits of which form a cascade. For a wide circuit, a good specification describes a “narrow” change of this circuit. In this report, we only give various “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 that 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 dramatically increases the search space. So an algorithm making such moves has no chance to converge to a better solution in reasonable time. For example, in a popular optimization method of simulated annealing (application of simulated annealing to logic synthesis is given in [2]), the number of moves increasing the cost function is controlled by the “cooling” scheduling. 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, \dots , (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 of gates in N^i . We will denote this number by $|N^i|$.) 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 the sequence gets stuck in a local minimum.

In this report, 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 report.) Let N be a single output circuit to be optimized and N_1, \dots, N_k be a partition of N into subcircuits. (In this report we assume, unless otherwise stated, that one needs to optimize a *single-output* circuit N .) 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 report, 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.

In [6], we introduced a generalization of LSPS of [3][4]. Let N_1, \dots, N_k be a specification of N . The generalization is to replace each subcircuit N_i , $i=1, \dots, k-1$ with subcircuit N_i^* whose toggling is implied by that of N_i . The subcircuit N_k is replaced with a toggle equivalent subcircuit N_k^* . This method of logic synthesis is more powerful than that of [3][4] because toggle equivalence is just a special case of toggle implication. For the sake of simplicity, in this report, we will use LSPS that preserves toggle equivalence (rather than toggle implication). Nevertheless everything that we say here about LSPS is applicable to the more general method of [6].

In this report, we also consider some other problems of LSPS. Namely we show that a narrow circuit N has a “natural” specification that is a cascade of subcircuits. On the other hand, if N is a wide circuit, its “good” specification can be viewed as a description of a “narrow change” of N .

This report 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 listed in Section 4. Section 5, describes application of LSPS to multi-output circuits. In Section 6, we relate LSPS to existing synthesis procedures from three different points of view. Section 7 analyzes LSPS from the optimization

point of view and shows that LSPS offers an elegant way to escape local minima. In Section 8 we introduce two types of optimization performed by LSPS: vertical and horizontal. Section 9 gives reasons for LSPS to be successful. In Section 10, we discuss finding good specifications for “narrow” and “wide” circuits. Finally, some conclusions are made in Section 11.

2. Example

Suppose that 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 subcircuits N_i of N are connected into an arbitrary directed acyclic graph.) The subcircuit N_1 implements the function $y = \text{square}(x)$ and N_2 implements the function $y < 100$.

It is not hard to see that the expression $x^2 < 100$ can be replaced with much simpler expression $\text{abs}(x) < 10$. 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.)

LSPS replaces N_1 with an optimized toggle equivalent subcircuit, e.g. with the subcircuit N_1^* implementing $y^* = \text{abs}(x)$. Then it computes relation $D_{\text{out}}(N_1, N_1^*)$ specifying the bijective mapping between the output assignments produced by subcircuits N_1 and N_1^* . (As it was shown 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 circuit $N_2^*(y^*)$ is constructed that is toggle equivalent to $N_2(y)$ (implementing $y < 100$) under the input constraint specified by relation $D_{\text{out}}(y, y^*)$. In the case N_1^* implements $y^* = \text{abs}(x)$, subcircuit N_2^* has to implement $y^* < 10$ (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).

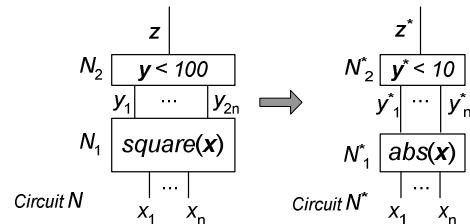


Figure 1. Optimization of $x^2 < 100$ by LSPS

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 $f: \{0,1\}^n \rightarrow \{0,1\}^m$ 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 two Boolean functions of the same set of variables. Functions f_1 and f_2 are called **toggle equivalent** if $f_1(x) \neq f_1(x') \Leftrightarrow f_2(x) \neq f_2(x')$. (Note that f_1 and f_2 may have different number of outputs.) Circuits N_1 and N_2 implementing toggle equivalent functions f_1 and 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 $f^* = f(\dots, x_i=0, \dots) \vee f(\dots, x_i=1, \dots)$.

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 $Sat(h)=1$ iff the assignment 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, $K^*(z_1, z_2) = 1$ iff $z_1 = K(z_2)$.

3.2 Correlation function

In this section, we use the notion of correlation function to extend definition of toggle equivalence to the case where functions f_1 and 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 $Cf(x, y)=1$ iff $x \in Q^X$ and $y \in Q^Y$ and $y = M(x)$.

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 $f_1(X)$ and $f_2(Y)$ be two multi-output Boolean functions where $X=\{x_1, \dots, x_k\}$ and $Y=\{y_1, \dots, y_p\}$ are sets

of their variables. (Note, that f_1 and f_2 may have different number of variables.). Let $Cf(X, Y)$ be a correlation function relating variables of f_1 and f_2 . Then one can introduce notions of toggle equivalence as follows. Boolean functions f_1 and f_2 are said to be toggle equivalent, if for any pair of pairs (x, y) and (x', y') of input vectors such that $Cf(x, y)=Cf(x', y')=1$, it is true that $f_1(x) \neq f_1(x') \Leftrightarrow f_2(y) \neq f_2(y')$.

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 $Sat(v(N_1)) \wedge Sat(v(N_2)) \wedge Cf(X, Y)$ 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, \dots, 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 $topological_level(N_i) \leq topological_level(N_j)$. In other words, subcircuits $N_i, i=1, \dots, k$ are processed by the LSPS procedure in topological order, from inputs to outputs.

```

1 LSPS( $N, Spec(N), cost\_function$ ) {
2   for ( $i=1; i <= k; i++$ ) {
3      $D_{imp}(N_i, N_i^*) = constraint\_function(N, N_i^*, i)$ ;
4      $N_i^* = synth\_toggle\_equivalent(N_i, D_{imp}, cost\_function)$ 
5      $D_{out}(N_i, N_i^*) = exist\_quantify(N_i, N_i^*, D_{imp})$ ; }
6 return( $N^*, Spec(N^*)$ )}

```

Figure 2. Pseudocode of LSPS procedure

Let us revisit the example of Section 2. LSPS starts with subcircuit N_1 (implementing $square(x)$) and recovers the function $D_{imp}(N_1, N_1^*)$ 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 $D_{imp}(N_1, N_1^*) \equiv 1$. 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 $D_{\text{out}}(N_1, N_1^*)$ is a correlation function.

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 $D_{\text{imp}}(N_2, N_2^*)$ relating inputs of N_2 and circuit N_2^* (synthesized in line 4) equals $D_{\text{out}}(N_1, N_1^*)$. (In general, the inputs of a subcircuit N_i of $\text{Spec}(N)$ are fed by outputs of more than one subcircuit N_j of $\text{Spec}(N)$. To obtain $D_{\text{imp}}(N_i, N_i^*)$ one has to take the conjunction of $D_{\text{out}}(N_j, N_j^*)$ for all subcircuits whose outputs feed inputs of N_i and N_i^* . It is not hard to show that in this case $D_{\text{imp}}(N_i, N_i^*)$ is a correlation function too.)

Let N_2^* be a subcircuit built by LSPS that is toggle equivalent to N_2 . If N_2^* is “irredundant” it has to have one output. (If, say a two-output circuit M' is toggle equivalent to a single-output circuit M , then either one output of M' is a constant or one output of M' is equal to the other output of M' or its negation.) 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].

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 specification $\text{Spec}(N)=\{N_1, \dots, N_k\}$ is the size of the largest subcircuit N_i of $\text{Spec}(N)$ (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 $\text{Spec}(N)$. So, if, for example, the size of subcircuits of $\text{Spec}(N)$ is bounded by a constant, the complexity of LSPS is linear.

The result above was improved in [5]. There, we considered the equivalence checking procedure for circuits N and N^* with a common specification (this procedure “enables” LSPS). We showed that the complexity of this equivalence checking procedure is exponential in the width of specifications $\text{Spec}(N)$ and $\text{Spec}(N^*)$ and linear in the number of subcircuits. The **width** of $\text{Spec}(N)$ is $\max(W_1, W_2)$. Here W_1 is the maximum number of outputs among the subcircuits N_i of $\text{Spec}(N)$ and W_2 is the maximum circuit width among the subcircuits N_i of $\text{Spec}(N)$. (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 $\text{Spec}(N)$ and $\text{Spec}(N^*)$ is not bounded (but the number of outputs and width of subcircuits of $\text{Spec}(N)$ and $\text{Spec}(N^*)$ is bounded). So the width of $\text{Spec}(N)$ provides a better parameterization of LSPS than its granularity.

4.2 Logic synthesis preserving toggle implication

In [6], we introduced a generalization of LSPS based on the notion of toggle implication. 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 $X=\{x_1, \dots, x_n\}$. 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 $f_2(x') \neq f_2(x'')$.

Let N be a single output circuit and $\text{Spec}(N)=\{N_1, \dots, N_k\}$. 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 “ \leq ” denotes the fact that toggling of N_i^* is implied by toggling of N_i .) The last subcircuit of $\text{Spec}(N)$ (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^*, \dots, N_k^* is functionally equivalent to N (modulo negation). In contrast to LS_TE, in LS_TI, when replacing subcircuit N_i , $i=1, \dots, k-1$ 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 toggling of such circuit is implied by toggling of N_i).

It is not hard to show (see [4]) 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 strictly more general relation, 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 The TEP procedure

The key part of LSPS is the procedure that, given a subcircuit N_i of $\text{Spec}(N)$, builds an optimized circuit N_i^* that is toggle equivalent to N_i (under input constraints specified by $D_{\text{imp}}(N_i, N_i^*)$). 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, \dots 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 .

5. Application of LSPS to multi-output circuits

In this section, we briefly discuss application of LSPS to optimization of multi-output circuits. Let N be a multi-output circuit. To generate a circuit N^* that is functionally equivalent to N we need a specification $Spec(N)$ such that every subcircuit N_i containing a primary output of N has only one output. An example of such a specification for a two-output circuit N is given in Figure 3. $Spec(N)$ consists of subcircuits N_1, N_2, N_3 where N_2 and N_3 are single-output subcircuits of N feeding its two primary outputs.

Suppose that circuits N_2 and N_3 share gates. Denote by $Gates(N)$ the set of gates of N . Since $Gates(N_2) \cap Gates(N_3) \neq \emptyset$, sets $Gates(N_1), Gates(N_2), Gates(N_3)$ form a cover of $Gates(N)$ rather than its partition.

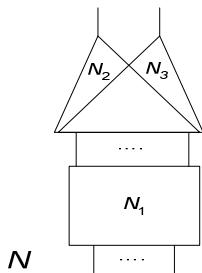


Figure 3. Specification of a two-output circuit

When formulating LSPS in [3] [4] [6], for the sake of simplicity we assumed that $Gates(N_i) \cap Gates(N_j) = \emptyset$ for two different subcircuits N_i, N_j of $Spec(N)$. However, the requirement can be easily relaxed. To handle the case of multi-output circuits, it is sufficient to require only that subcircuits N_i of $Spec(N)$ do not share “output gates”. (That is a gate of N_i whose output is an output of N_i can not be in another circuit N_j . However, N_i and N_j may share “internal” gates.) However, one can relax the definition of “permissible” specification even more. For example, one can have a specification $Spec(N)$ where output gates of N_i and N_j are shared. As long as $Spec(N)$ satisfies the two conditions below:

- one can build the graph G (see Subsection 3.3) describing connections between subcircuits of $Spec(N)$;
- the graph G is acyclic

one can apply LSPS. So one, for example, should avoid partitions where an output gate of N_i is an internal node of N_j (because it is not clear how to build G in such a case).

It is not hard to see that by replacing subcircuits $N_i, i=1,2,3$ shown in Figure 3 with toggle equivalent subcircuits N_i^* , LSPS produces a circuit N^* that is functionally equivalent to N modulo negation of outputs. (For single-output circuits N_2^* and N_3^* toggle equivalence

with N_2 and N_3 means functional equivalence modulo negation.) To minimize the size of N^* one should try to make N_2^* and N_3^* share as much logic as possible. Suppose circuit N_2^* is synthesized before N_3^* . Then when synthesizing N_3^* , the logic of N_2^* may be re-used. This can be done by slightly modifying the TEP procedure mentioned in Subsection 4.3. However, the discussion of this topic is beyond the scope of this report.

6. Relation of LSPS to existing synthesis methods

In this section, we relate LSPS to other methods of logic synthesis from three different points of view. Since we give a very high-level comparison, we do not reference the existing methods of logic synthesis. (A comparison of LSPS with SPFDs [8][9] can be found in [7].)

6.1 Comparison in terms of enabling equivalence checking procedures

In this subsection, we consider the difference between LSPS and existing logic synthesis procedures from the viewpoint of enabling equivalence checking procedures.

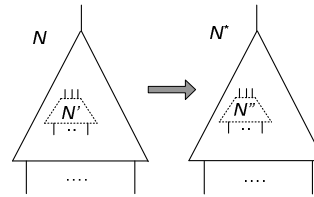


Figure 4. A typical synthesis transformation

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 4, 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 two 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 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 $D_{\text{out}}(N', N'')$. 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 logic fed by N' to make the entire transformation correct.

6.2 Comparison in terms of handling complexity

In this subsection, we compare existing methods of logic synthesis with LSPS from the viewpoint of complexity handling. A typical transformation performed by a logic synthesis procedure is shown in Figure 4. Verification of correctness of this transformation is simplified by a) limiting the size (or width) of subcircuit N' ; b) making the optimized circuit N'' functionally equivalent to N' . By limiting the size of N' one simplifies the “block-level part” of verification i.e. checking that subcircuits N' and N'' are functionally equivalent. By making N'' functionally equivalent to N' one limits the *scope of transformation* and so trivializes the compositional part of verification.

LSPS reduces its complexity by using $\text{Spec}(N)$ that consists of “narrow” subcircuits N_i with a bounded number of outputs (Subsection 4.1). This lowers the complexity of replacing N_i with a toggle equivalent counterpart N_i^* and the complexity of computing $D_{\text{out}}(N_i, N_i^*)$. However, LSPS remains scalable *without scoping* (since N_i^* is not functionally equivalent to N_i , the replacement of N_i with N_i^* affects the logic fed by N_i). So, to keep LSPS scalable, it is sufficient to use specification $\text{Spec}(N)$ of small width. This improvement in complexity handling is due to the progress in equivalence checking made in [4][5]. Previously, the formal results on equivalence checking of circuits N and N^* performed either by BDDs or by SAT were expressed in terms of *absolute complexity* of N and N^* . For example, if N and N^* have small width then their equivalence can be efficiently established by building their BDDs. The complexity of equivalence checking of circuits with a common specification is formulated *in relative terms*. If N

and N^* have a “narrow” common specification then no matter how complex (or wide) circuits N and N^* are, they can be checked for equivalence efficiently (if this common specification is known). The fact that $\text{Spec}(N)$ of N is narrow means that we make a “narrow” change of N (but in contrast to existing methods of synthesis this change may encompass the entire circuit).

6.3 Comparison in terms of “friendliness” of environment

LSPS introduces a new type of subcircuit/environment interaction. Let us consider again the transformation shown in Figure 4 where subcircuit N' is replaced with functionally equivalent subcircuit N'' . If one considers $N \setminus N'$ as the “environment” of subcircuit N' , then this environment can be called “unfriendly”. Indeed, if N'' is not functionally equivalent to N' , the environment “punishes” this transformation by making the resulting circuit incorrect. In LSPS, one can replace subcircuit N' with a *toggle equivalent* counterpart N'' . Since toggle equivalence means re-encoding, the logic fed by the outputs of N'' has to change. This is done by computing correlation functions and replacing subcircuits of N with toggle equivalent counterparts as described before. The replacement of N' with toggle equivalent subcircuit N'' is possible because the “environment” $N \setminus N'$ “cooperates” with N' by making changes in the surrounding logic in such a way that the replacement of N' with N'' become “unobservable”. Such cooperation allows one to explore a much richer space of transformations.

7. 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 $\text{Spec}(N)$. The deeper a local minimum is, the more coarse partitioning of N into subcircuits is necessary to avoid it. In particular, if $\text{Spec}(N)$ consists of N itself, then LSPS can 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 7.1 we recall the problem of local minima entrapment in the context of traditional logic synthesis.

Subsection 7.2 describes a modification of LSPS called LSPS⁺. Since LSPS is a special case of LSPS⁺, then everything we say about LSPS⁺ applies to LSPS as well. Subsection 7.3 shows that LSPS⁺ can escape local minima that trap solutions of traditional synthesis methods.

7.1 Local minima entrapment

Let N be a circuit to be optimized. A typical synthesis procedure performs a sequence of transformations shown in Figure 4. Each transformation reduces the value of a cost function (as we mentioned above, in this report we assume that $cost(N)=|N|$). Then a typical synthesis procedure builds a sequence of circuits N^1, N^2, \dots , 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.

7.2 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 5. On the one hand, we use LSPS⁺ to explain what LSPS is 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*.

```

1 LSPS+( $N, Spec(N), cost\_function$ ) {
2   for ( $i=1; i <= k; i++$ ) {
3      $D_{inp}(N_i, N_i^*) = constraint\_function(N, N_i^*, i)$ ;
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  $|R_i^*| = \infty$ 
8     if ( $(|N_i^*| + .. + |N_i^*| + |R_{p1}^*| + .. + |R_{pi}^*|) < |N_i| + .. + |N_i|$ )
9       return( $N^*, Spec(N^*), R_{p1}^*, \dots, R_{pi}^*$ );
10  return( $N^*, Spec(N^*)$ )}

```

Figure 5. Pseudocode of LSPS⁺

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 “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 .

Let us explain how LSPS⁺ works by the example shown in Figure 6 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 5) such that $R_1^*(N_1^*(y))$ is *functionally equivalent* to $N_1(y)$. (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_2^*, R_1^* and N_2 .

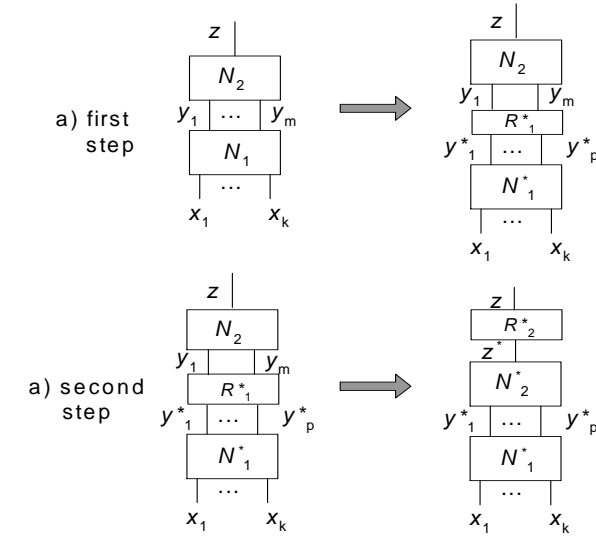


Figure 6. Example of LSPS⁺ run

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_2^* if the latter is the negation of N . (Note that at this point circuit R_1^* “disappears” from the circuit. For that reason, in line 8 of Figure 6 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 $R_{p1}^*, \dots, R_{pi}^*$ 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 1. 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 $LSPS$.

7.3 Escaping local minima by $LSPS^+$

Suppose that during the run of $LSPS^+$ shown in Figure 6, 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 $|N^*| + |R_1^*| \geq |N_1|$, eventually it managed to build a circuit N^* smaller than N . Inequality $|N^*| + |R_1^*| \geq |N_1|$ 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_1^* such that $|N^*| + |R_1^*| < |N_1|$, the re-encoder R_1^* built by $LSPS^+$ is larger than R_1^* and so $|N^*| + |R_1^*| \geq |N_1|$. Third, there is no re-encoder R_1^* such that $|N^*| + |R_1^*| < |N_1|$. 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.

8. Horizontal and vertical optimization

In Subsections 8.1 and 8.2 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 .)

8.1 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**.

An example of horizontal optimization is shown in Figure 7. 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 $y = square(x)$, $z = 3 * square(x)$ and $sum(y, z)$ 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 $z = mult(3, y)$. 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.

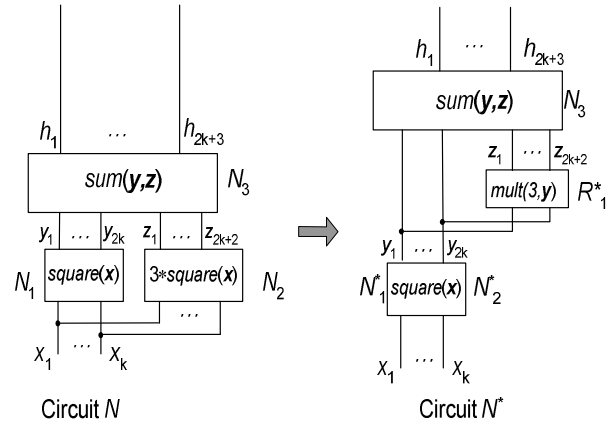


Figure 7. Example of horizontal optimization

8.2 Vertical optimization

Let us return to the example of Section 2. Application of $LSPS^+$ to this example is shown in Figure 8. $LSPS^+$ performs two steps. In the first step, the subcircuit N_1 implementing $square(x)$ is replaced with circuit N_1^* implementing $abs(x)$ and re-encoder R_1^* . In the second step, re-encoder R_1^* and circuit N_2 (implementing $y < 100$) 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 $N_2(R_1^*(y^*))$.

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 $R_1^*(N_1^*(x))$ is equivalent to $N_1(x)$ 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.) 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**.

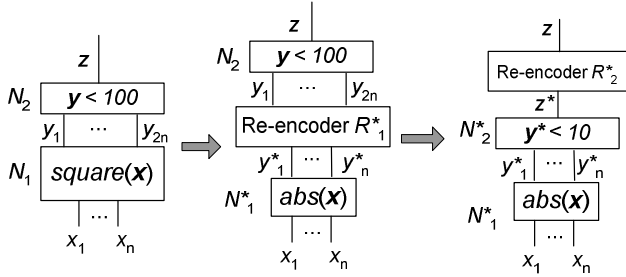


Figure 8. Vertical optimization by LSPS

9. Why should it work?

In this section, we discuss the reasons for $LSPS^+$ to succeed in circuit optimization. In Subsection 9.1, we show that $LSPS^+$ provides a framework for designing efficient algorithms that can escape local minima. In the following subsections we give various aspects of $LSPS^+$ that should make it successful. In Subsection 9.2, we show that horizontal optimization is a natural way to share logic between “cooperating” logic blocks. Subsections 9.3 and 9.4 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 9.3). However, even if N does not lose information globally or loses very “little”, it still can have subcircuits that lose information locally (Subsection 9.4).

9.1 High-level view

$LSPS^+$ can be viewed as just a framework for studying and designing algorithms that can escape local minima. Suppose we try to optimize a circuit N using a set of small equivalent transformations as shown in Figure 4. 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.

9.2 Horizontal optimization

Before, we showed a *made-up* example of applying horizontal optimization successfully (Figure 7). 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 report.)

9.3 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, \dots, 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 x, y be a pair of input vectors such that $x \neq y$ and $N(x) = N(y)$. Then there should be a cut

$C_i, i=2, \dots, p$ such that $C_i(\mathbf{x})=C_i(\mathbf{y})$ and for every cut $C_j, j > i$ it is also true that $C_j(\mathbf{x})=C_j(\mathbf{y})$. 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 $square(\mathbf{x})$ with N_1^* implementing $abs(\mathbf{x})$, $LSPS^+$ runs up a large “debt”. However, since the circuit N (namely its subcircuit N_2 implementing $y < 100$) 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 $y' < 10$) $LSPS^+$ pays only a small fraction of this debt and nevertheless obtains circuit N^* functionally equivalent to N .

9.4 Vertical optimization (local loss of information)

Let N be a circuit to be 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 9.3, $LSPS^+$ 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.

10. Finding specification

In this section, we consider various aspects of finding a “good” specification of a circuit. Informally, a specification $Spec(N)=\{N_1, \dots, N_k\}$ is good, if it reflects a natural flow of information.. In Subsection 10.1, we discuss whether a good specification can be found automatically and conjecture that, in general, it is hard if not impossible. Subsection 10.2 shows, however, that for circuits of small width there is a very simple natural specification. This specification is a cascade of subcircuits. So in case of narrow circuits, finding a good specification automatically is possible. This is an important fact because narrow circuits are ubiquitous in real-life designs. Finally, in Subsection 10.3, we consider the case of wide circuits. One can not just take as a specification a “natural” partitioning of a wide circuit N into subcircuits N_1, \dots, N_k . The width of such specification may be large. However, a specification of small width can be *derived* from a natural specification. Such a specification describes a narrow change of N .

10.1 Finding specification automatically

In this subsection, we discuss whether one can find a good specification automatically. Intuitively, $Spec(N) = \{N_1, \dots, N_k\}$ is a good specification if, when one output of N_i feeds an input of N_j , then all (or “almost all”) outputs of N_i feed inputs of N_j . Otherwise, one can have the situation shown in Figure 9. This figure depicts a fragment of a circuit N (on the left) consisting of subcircuits N_1, \dots, N_4 . Note that only one out of three outputs of N_1 feed $N_i, i=2,3,4$. (For example, the output a feeds one input of N_2 .) The result of application of $LSPS^+$ is shown on the right. After replacing subcircuits N_1, \dots, N_4 with their toggle equivalent counterparts, all three outputs of N_1^* feed each subcircuit $N_i^*, i=2,3,4$. (The connections are shown only for N_2^* .) So, in this case, $LSPS^+$ results in artificially increasing the information flow between subcircuit N_1^* and subcircuits N_2^*, N_3^*, N_4^* .

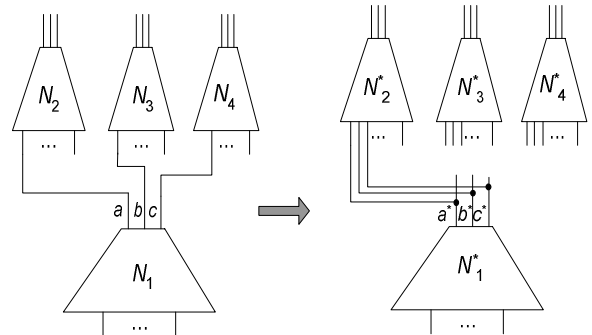


Figure 9. An example of “poor” specification

Finding a good specification automatically is, in general, hard if not impossible. There are two reasons for that. First, the number of potential subcircuits is exponential

in subcircuit size, which makes their enumeration infeasible. Second, there is a high probability of a “false positive”. Suppose, for example, that all three outputs of N_1 of Figure 9 feed each of the subcircuits N_2, \dots, N_4 . However, this may not be true for subcircuits N_2, \dots, N_4 themselves. So even though the choice of N_1 originally seemed to be reasonable, later, one may discover that it was a mistake.

10.2 Finding specification for “narrow” circuits

In the previous subsection, we conjectured that finding a good specification, is probably infeasible, in general. Nevertheless, there is a very important class of circuits that have a “trivial” specification. These are circuits of small width. Due to triviality of their specification, a reasonably good specification can be found efficiently. An example of a narrow circuit is shown in Figure 10 on the left. Primary inputs and gates of a narrow circuit N can be ordered in a such way (we consider here only topological orderings) that N can be covered by a “long” and “narrow” box. The size of a horizontal cut C of such a box is small. This cut consists of variables describing gate outputs and primary input variables (in case a primary input variable is located below C and feeds a gate above C .)

A narrow circuit N has a “natural” specification N_1, \dots, N_k shown in Figure 10 on the right. Each subcircuit $N_i, i=2, \dots, k$ has output variables $z_i^1, \dots, z_i^{d_i}$ and input variables $x_i^1, \dots, x_i^{p_i}$ and $z_{i-1}^1, \dots, z_{i-1}^{d_{i-1}}$. Here $x_i^1, \dots, x_i^{p_i}$ are primary input variables of N . We assume that sets of input variables of N_i and $N_j, i \neq j$ do not overlap. If an input variable x_i^m of N_i feeds two gates that are located in subcircuits N_i and N_j of $Spec(N)$ where $j > i$, then some variable z_i^s of N_i is equal to x_i^m .

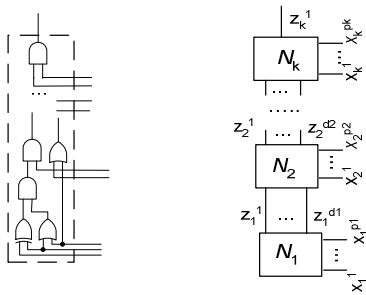


Figure 10. Specification of a “narrow” circuit

Since N is a narrow circuit, there is a topological ordering of variables of N such that the size of the number of outputs for every N_i is bounded by a “small” constant. Besides, since N is a narrow circuit, subcircuits N_i are also narrow. So a natural specification of N has a small width.

Since the topology of a natural specification of a narrow circuit is known, one has a good chance to find a high-

quality specification automatically. First, one needs to find a good topological ordering for the variables of N . Then N is “sliced” into a cascade of subcircuits N_i of manageable size.

10.3 Finding specification for “wide” circuits

Let N be a wide circuit. Such a circuit should have a “natural” specification consisting of narrow subcircuits N_1, \dots, N_k . (The reason is that building a structureless wide circuit that performs a meaningful computation is hard if not infeasible.) However, one can not use subcircuits N_1, \dots, N_k as a specification of N because they may have an unbounded number of outputs (then specification $\{N_1, \dots, N_k\}$ has a large width even if subcircuits N_i are narrow). In such a case, one can partition N_i further into subcircuits of small number of outputs. This way one can build a specification $Spec(N)$ extracted from a natural specification $\{N_1, \dots, N_k\}$.

Let us illustrate the said above by the example of a multiplier that is a “classic” wide circuit. Let us consider a trivial implementation N of a multiplier as a cascade of regular adders. Then N has a natural specification which is a partition of N into subcircuits N_i representing adders. (An adder is a narrow circuit). However this partition cannot be used by $LSPS^+$ because each adder has a larger number of outputs. To solve this problem, one can partition each N_i into subcircuits N_i^p where N_i^p implements the adjacent outputs of the adder N_i . (For example, subcircuit N_i^1 implements the s least significant output bits of the adder N_i , subcircuit N_i^2 implements the next s bits of N_i and so on.)

The set of subcircuits N_i^p forms a narrow specification $Spec(N)$ of the multiplier N that can be used by $LSPS^+$. It is not hard to see that $Spec(N)$ is a good specification in terms of Subsection 10.1. Although it is unlikely to get a smaller multiplier by applying $LSPS^+$ to N with specification $Spec(N)$, one can try to improve the performance of this multiplier. A regular adder is a deep circuit and so it is slow. For that reason, various schemes have been designed to improve adder’s performance. $LSPS^+$ can try to achieve the same goal by replacing subcircuits N_i^p with more shallow toggle equivalent counterparts N_i^{*p} .

One can say that $Spec(N)$ above describes a “narrow change” of the multiplier N . Importantly, $Spec(N)$ is not identical to a natural high-level specification of N but is derived from it.

11. Conclusions

In this report, 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. We also discuss the problem of finding a good specification of a circuit. Namely, we show that narrow circuits have a very simple natural specification which is a cascade of subcircuits. For a wide circuit, a good specification can be extracted from a natural partitioning of this circuit into narrow subcircuits.

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] S.Sinha, R.K.Brayton. *Implementation and use of SPFDs in optimizing Boolean networks*. ICCAD-1998, pp. 103-110.
- [9] S.Yamashita,H.Sawada,A.Nagoya. *A new method to express functional permissibiities for LUT based FPGAs and its applications*.ICCAD-1996,pp.254-261.